Younes Kabiri Farah

DISEÑO, IMPLEMENTACIÓN Y DESPLIEGUE DE UN SISTEMA DE PRESUPUESTACIÓN Y UNA APLICACIÓN ADMINISTRATIVA PARA UNA EMPRESA INMOBILIARIA

TRABAJO DE FINAL DE GRADO

dirigido por Dr. Sánchez Artigas, Marc

Grado en Ingeniería Informática

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

RESUMEN: En la última década, el sector de las Start-ups ha experimentado un crecimiento exponencial en España, impulsado en parte por el Plan España Digital 2025, que busca la transformación digital de las empresas. En nuestro caso, la empresa REFORMA se ha puesto en contacto con nosotros con el fin de digitalizar y automatizar su proceso de trabajo, el cual, anteriormente, se llevaba a cabo de manera manual y presencial. Esta manera de trabajar supone un coste adicional debido que un porcentaje de usuarios no continua con la contratación después de dicha inversión, este se puede reducir mediante la implementación de un sistema. Este les permite gestionar proyectos de manera más eficiente, satisfaciendo mejor a los clientes y reduciendo costos.

RESUM: En l'última dècada, el sector de les Start-ups ha experimentat un creixement exponencial a Espanya, impulsat en part pel Pla Espanya Digital 2025, que busca la transformació digital de les empreses. En el nostre cas, l'empresa REFORMA s'ha posat en contacte amb nosaltres amb la finalitat de digitalitzar i automatitzar el seu procés de treball, el qual, anteriorment, es duia a terme de manera manual i presencial. Aquesta manera de treballar suposa un cost addicional, ja que un percentatge d'usuaris no continua amb la contractació després d'aquesta inversió, cosa que es pot reduir mitjançant la implementació d'un sistema. Aquest, els permet gestionar projectes de manera més eficient, satisfent millor als clients i reduint costos.

ABSTRACT: In the last decade, the Start-up sector has experienced exponential growth in Spain, promoted in part by the Spain Digital 2025 Plan, which focuses on the digital transformation of companies. In our case, the company REFORMA has contacted us in order to digitize and automatize its work process, which, previously, was carried out manually and in person. This way of working involves an additional cost because a percentage of users do not continue with the contracting after this investment, this can be reduced by implementing a system. This allows them to manage projects more efficiently, better satisfying customers and reducing costs.

Tabla de Contenido

1. INTRODUCCIÓN **9**

1.1. OBJECTIVOS 11
1.1.1. Objetivos Principales *11*
1.1.2. Objetivos Secundarios *11*
1.2. MOTIVACIÓN 12
1.3. PLANIFICACIÓN 13
1.4. ORGANIZACIÓN DE LA MEMORIA 15

2. BACKGROUND **16**

2.1. INTRODUCCIÓN AL DESARROLLO WEB 16
2.1.1. Lenguajes de Programación Web *16*
2.1.2. Frameworks y bibliotecas *17*
2.2. BASES DE DATOS 18
2.2.1. Bases de datos relacionales *18*
2.2.1.1. Ejemplos de Bases de datos relacionales 20
2.2.2. Bases de datos NoSQL *21*
2.2.2.1. Características 21
2.2.2.2. Ejemplos Bases de datos NoSQL 21
2.2.3. Otras bases de datos *22*
2.2.3.1. Bases de datos de objetos 22
2.2.3.2. Bases de datos en memoria 22
2.2.3.3. Bases de datos especiales 22
2.2.3.4. Bases de datos temporales 22
2.3. EXPERIENCIA DE USUARIO [10] 23

3. DISEÑO DE LA PROPUESTA: **24**

3.1. REQUISITOS [11] 26
3.1.1. Requisitos Funcionales *26*
3.1.2. Requisitos No Funcionales *28*
3.2. DIAGRAMAS UML 29
3.2.1. Sistema presupuestación *29*
3.2.2. Aplicación administrativa *31*
3.2.2.1. Rol administrativo 31
3.2.2.2. Rol cliente 33
3.2.2.3. Rol trabajador 34
3.2.3. Diseño de la base de datos *35*
3.2.3.1. Diseño del sistema de presupuestación 35
3.2.3.2. Diseño del sistema de la aplicación administrativa 36

4. IMPLEMENTACIÓN [12] **37**

4.1. ELECCIÓN DE LAS TECNOLOGÍAS 38
4.1.1. Tecnologías Front-end *38*
4.1.2. Tecnologías Back-end *40*
4.1.2.1. TypeScript 40
4.1.2.2. TypeORM 40
4.1.2.3. Cloud computing: 41
4.1.2.4. Docker 43
4.2. IMPLEMENTACIÓN BACK-END APLICACIÓN ADMINISTRATIVA 45
4.2.1. Paquetes instalados *45*
4.2.2. Estructura del proyecto *46*
4.2.3. Funcionalidad *47*
4.2.3.1. App.ts 47
4.2.3.2. Rutas 49
4.2.3.3. Auth 50
4.2.3.4. Entidad usuario 51
4.2.3.5. Login 52

4.2.3.6. Multer 54
4.2.3.7. Amazon S3 55
4.2.3.8. Amazon SES 58
4.2.3.9. Guard 63
4.2.3.10. Relaciones entre entidades 65
4.2.3.11. Documentación 67
4.2.3.12. Dockerfile 71
4.2.3.13. Logger 73
4.3. IMPLEMENTACIÓN BACK-END SISTEMA PRESUPUESTACIÓN 80
4.3.1. Amazon DocumentDB 80
4.3.2. Datos del Presupuestador 81
4.3.3. Almacenamiento de datos del Presupuestador 82
4.3.4. Generación de PDF 82
4.3.5. Envío de correo 84
4.4. IMPLEMENTACIÓN FRONT-END APLICACIÓN ADMINISTRATIVA 86
4.4.1. Componentes fundamentales de Angular [33][34] 86
4.4.2. Estructura del proyecto 88
4.4.3. Ficheros globales 89
4.4.4. Funcionalidad 90
4.4.4.1. Auth 90
4.4.4.2. Rutas 94
4.4.4.3. Guards 98
4.4.4.4. Servicios 100
4.4.4.5. Modelos 101
4.4.4.6. Pipes 102
4.4.4.7. Pop up 103
4.4.4.8. Vistas aplicación administrativa 105
En este apartado vemos algunas vistas de ejemplo de la aplicación administrativa: 105
4.5. IMPLEMENTACIÓN FRONT-END SISTEMA PRESUPUESTACIÓN 107
4.5.1. Sistema de presupuestación 107
4.5.2. Componente inicio Landing Page. 108
4.5.3. Componente inicio Presupuestador. 109
4.5.4. Componente reforma-integral 111

5. EVALUACIÓN 115

6. DESPLIEGUE 118

3.1. CREACIÓN DE BASES DE DATOS 119
3.1.1. Base de datos del sistema administrador - Amazon RDS 119
3.1.2. Base de datos sistema presupuestación 121
3.2. AMAZON FARGATE 122
3.2.1. Security Groups 123
3.2.2. Target Groups 124
3.2.3. Load Balancer 125
3.2.4. Amazon Elastic Container Registery (ECR) 127
3.2.5. Asignar recursos a Fargate 130

7. CONCLUSIONES 131

8. REFERENCIAS 132

Abreviaturas

RDBMS: Sistema de gestión de bases de datos relacionales.
SQL: Las siglas en inglés Structured Query Language.
GDPR: Reglamento General de Protección de Datos en la Unión Europea.
DOM: Document Object Model.
AWS: Amazon Web Services.
JSON: JavaScript Object Notation.
JWT: JSON Web Tokens.
ORM: Object Relational Mapping o Mapeo Objeto-Relacional.
DNS: Domain Name System o Sistema de nombres de dominio.
VPC: Virtual Private Cloud.
BLOB: Binary Large Object.
SDK: Software Development Kit.
YAML: YAML Ain't Markup Language.

Índice de figuras

Figura 1. Ciclo de la reforma. Elaboración Propia. 9
Figura 2. Diseño del sistema a implementar. Elaboración Propia. 10
Figura 3. Diagrama de Gantt. Elaboración Propia. 14
Figura 4. Desglose comparado por subregión: https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fg%2F11c6w0ddw9,%2Fm%2F012l1vxv,%2Fg%2F11c0vmgx5d 17
Figura 5. Interés entre Angular, React y Vue.js. https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fg%2F11c6w0ddw9,%2Fm%2F012l1vxv,%2Fg%2F11c0vmgx5d 17
Figura 6. Interés entre Django, Ruby on Rails y node.js: https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fm%2F06ff5,%2Fm%2F06y_qx,%2Fm%2F0bbxf89....17
Figura 7. Diagrama ACID. Elaboración Propia. 18
Figura 8. Atomicitiy – Ejemplo operación bancaria. Elaboración Propia. 19
Figura 9. Consistency – Ejemplo operación bancaria. Elaboración Propia. 19
Figura 10. Diagrama relacional: https://www.lucidchart.com/pages/es/ejemplos/herramienta-ERD....20
Figura 11. https://www.scylladb.com/glossary/nosql-design-principles/ 21
Figura 12. Ciclo de la reforma. Elaboración Propia. 24
Figura 13. Diseño del sistema a implementar. Elaboración Propia. 25
Figura 14. Proceso para generar el presupuesto. Elaboración Propia....26
Figura 15. Diagrama de caso de uso del sistema de presupuestación. Elaboración Propia....29
Figura 16. Diagrama de frecuencia del sistema de presupuestación. Elaboración Propia....30
Figura 17. Diagrama de caso de uso del administrador. Elaboración Propia 31
Figura 18. Diagrama de caso de uso del cliente. Elaboración Propia....33
Figura 19. Diagrama de caso de uso del cliente. Elaboración Propia....34
Figura 20. Diagrama de clases del sistema de presupuestación. Elaboración Propia....35
Figura 21. Diagrama de clases de la aplicación administrativa. Elaboración Propia 36
Figura 23. Diagrama sistema de presupuestación final. 37
Figura 22. Diagrama sistema administrativo final. 37
Figura 24. Logo de Angular: https://es.wikipedia.org/wiki/Angular_%28framework%29 ...38
Figura 25. Logo de React: https://commons.wikimedia.org/wiki/File:React-icon.svg 39
Figura 26. Logo de TypeScript: https://blog.toothpickapp.com/top-10-things-to-know-about-typescript/....40
Figura 27. Figura de los tres competidores Cloud: https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fm%2F04y7lrx,%2Fm%2F0105pbj4,%2Fm%2F0rznzt1 41
Figura 28. Logo de AWS cloud: https://unaaldia.hispasec.com/2020/08/descubierto-criptominer-integrado-en-amazon-aws-community-ami.html....41
Figura 29. Figura de los servicios de AWS: https://www.projectpro.io/article/aws-projects-ideas-for-beginners/453 42
Figura 30. Figura de servidores Bare-metal. Contenido del curso de AWS developer. 43
Figura 31. Figura de servidores Bare-metal. Contenido del curso de AWS developer. 43
Figura 32. Figura de servidores Bare-metal. Contenido del curso de AWS developer. 44
Figura 33. Logo de Docker: https://inlab.fib.upc.edu/ca/blog/docker-devops-tool. 44
Figura 34. Diagrama de la autenticación. Elaboración Propia 50
Figura 35. Correo recibido por el cliente. 85
Figura 37. Vista trabajadores - Rol Administrador....105

Figura 36.Vista calendario - Rol Administrador.....105
Figura 39. Vista cliente - Rol Administrador. 106
Figura 40. Vista obra - Rol Administrador. 106
Figura 38. Vista imágenes - Rol Administrador. 106
Figura 41. Vista obra - Rol Administrador. 106
Figura 42. Página inicio del landing page.....108
Figura 43. Vista ordenador MacBook-Pro.....109
Figura 45. Vista iPad 11. 110
Figura 44. Vista iPhone 14. 110
Figura 47. Vista ordenador MacBook-Pro del Componente reforma-integral 1. 111
Figura 46. Vista ordenador MacBook-Pro del Componente reforma-integral 2. 111
Figura 48. Vista iPad 11 del Componente reforma-integral. 112
Figura 49. Vista iPhone 14 del Componente reforma-integral. 112
Figura 50. Vista iPad 11 del Componente reforma-integral - precio.....113
Figura 51. Vista iPhone 14 del Componente reforma-integral - precio.....113
Figura 52. Vista del formulario para solicitar el documento PDF.....114
Figura 53. Figura AWS Fargate.....118
Figura 54. Inicio sesión AWS CLI. 128

Índice de tablas

Tabla 1. Tareas del sistema de presupuestación. Elaboración Propia.....26
Tabla 2. Tareas de la aplicación administrativa rol Administrador. Elaboración Propia. 27
Tabla 3. Tareas de la aplicación administrativa rol Cliente. Elaboración Propia. 27
Tabla 4. Tareas de la aplicación administrativa rol Trabajador. Elaboración Propia. 27
Tabla 5. Tareas de la aplicación administrativa rol Trabajador. Elaboración Propia. 28
Tabla 6. Evaluación tareas del sistema presupuestación. Elaboración propia. 115
Tabla 7. Evaluación tareas de la aplicación administrativa rol Administrador. Elaboración Propia. 116
Tabla 8. Evaluación tareas de la aplicación administrativa rol Trabajador. Elaboración Propia. 116
Tabla 9. Evaluación tareas de la aplicación administrativa rol Cliente. Elaboración Propia. 116
Tabla 10. Evaluación tareas de la aplicación administrativa rol Trabajador. Elaboración Propia. 117

1. Introducción

En la última década, el sector Start-up ha experimentado un crecimiento exponencial tanto a nivel territorial como en el resto de mundo [39]. Uno de los factores que ha promovido el gobierno español para la creación de estas nuevas empresas (o bien para la modernización de las ya existentes), es el Plan España Digital 2025[1] presentado el 23 de julio 2020 [40]. Este plan tiene como objetivo impulsar la transformación digital de las empresas a fin de relanzar el crecimiento económico, la reducción de la desigualdad, aumentar la productividad, y sobre todo, aprovechar todas aquellas ventajas y oportunidades que ofrecen las nuevas tecnologías.

REFORMA es una empresa emergente que se dedica a brindar servicios de reformas y renovación domiciliaria. Esta empresa ha sido impulsada también por el plan España Digital con la intención de digitalizar sus servicios. Hasta ahora, solamente han dispuesto de una página web estática que brindaba información de contacto a los posibles clientes para proceder con la reforma. De tal manera, esta página era el único elemento digital del que disponían, puesto que el resto se llevaba a cabo de forma personal y manual.

En la **Error! Reference source not found.** mostramos el proceso tradicional que sigue la empresa desde el primer contacto con el cliente hasta la finalización de la reforma. Como podemos observar, el proceso consta de varias etapas, que hasta ahora se han realizado de forma presencial y manual. Esto ha supuesto a la empresa gastos tanto temporales como económicos, los cuales no son rembolsable si el cliente no continua con la reforma. Dichos costes se pueden reducir gracias a la digitalización mediante un sistema que pueda automatizar algunas de estas etapas sin necesidad de intervención humana.

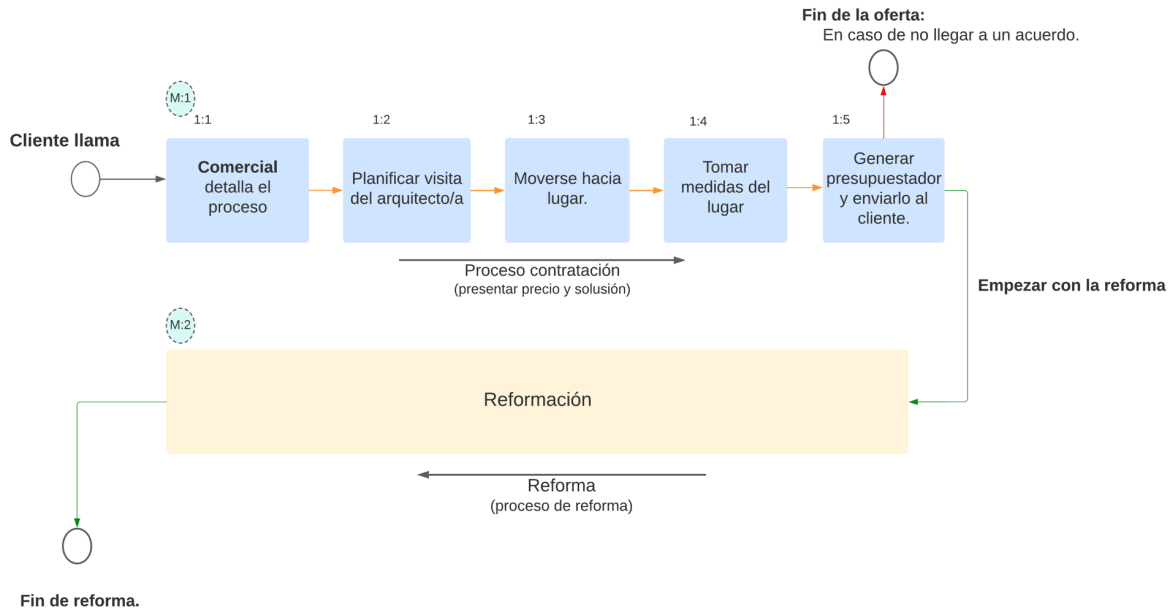


***Figura 1**. Ciclo de la reforma. Elaboración Propia.*

[1] https://sede.red.gob.es/es/procedimientos/convocatoria-de-ayudas-destinadas-la-digitalizacion-de-empresas-del-segmento-i-entre

El siguiente proyecto tiene como objetivo principal optimizar las operaciones y mejorar la capacidad de gestión de esta empresa. Para ello, implementamos un software integral formado por un presupuestador y un sistema de gestión de reformas como podéis visualizar en la Figura 2.

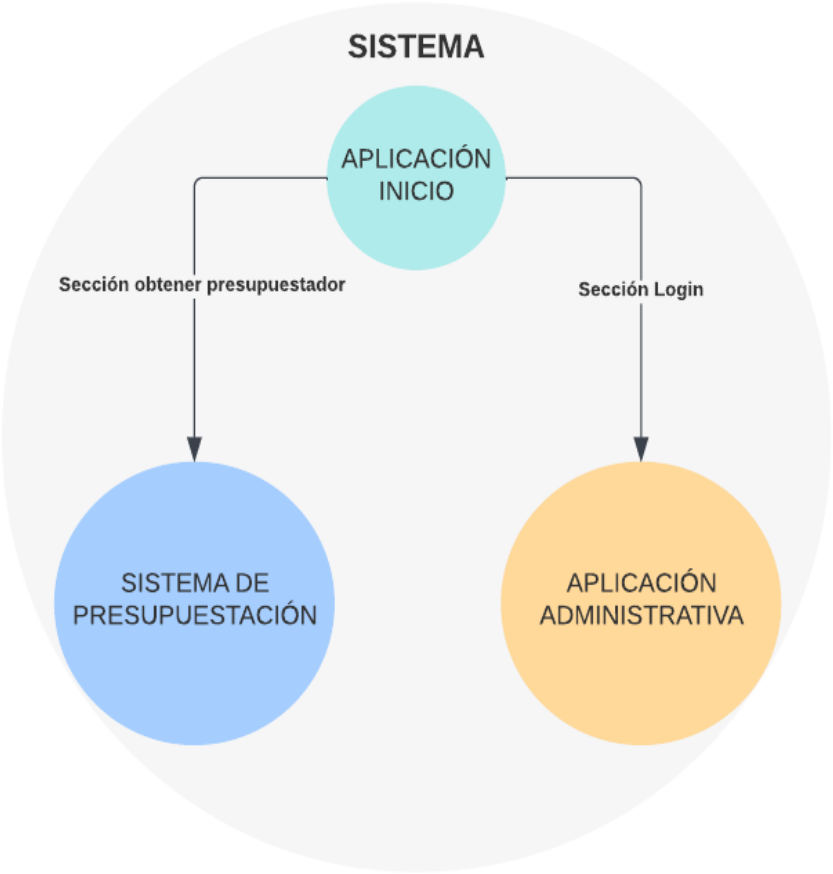


***Figura 2**. Diseño del sistema a implementar. Elaboración Propia.*

Gracias a este software, la empresa puede optimizar el desarrollo del ciclo de las reformas, lo cual le permite aceptar y gestionar más proyectos. Por otro lado, el cliente estará más satisfecho, ya que mediante el mismo software podrá observar la evolución de su reforma, además de tener una comunicación directa con la empresa.

1.1. Objectivos

La empresa REFORMA necesita un sistema que le ayude a brindar un presupuesto al cliente de manera rápida y que no requiera de muchos pasos. Además de esta funcionalidad, también piden un sistema que les ayude a gestionar los diferentes aspectos de las reformas, partiendo de la gestión de la plantilla, pasando por los eventos e incidencias y hasta la agilización de la comunicación entre la empresa y los clientes.

Para llevar a cabo un proyecto de este tamaño, lo dividimos en dos sistemas: el sistema de presupuestación y el sistema administrativo.

1.1.1. Objetivos Principales

La empresa nos ha solicitado tanto el diseño, la implementación como el despliegue de los diferentes sistemas y como es de esperar, este proceso se tiene que realizar de la manera más eficiente y ágil.

Los objetivos a cumplir son:

1. Diseñar e implementar un presupuestador que, dadas unas medidas, devuelva el coste de la reforma de manera rápida y exacta.
    a. *Aplicación web para introducir medidas, obtener y solicitar el presupuesto en forma de documento PDF.*
    b. *Back-end para obtener los precios y porcentajes para realizar el cálculo. También debe almacenar tanto los datos que introduce el cliente en la web como los que esta misma ofrece.*
2. Diseñar e implementar un sistema que facilite las tareas diarias a los miembros de la empresa.
    a. **Administradores**:
        i. realizar tareas administrativas.
    b. **Trabajadores**:
        i. Poder visualizar diferentes eventos (tareas).
        ii. Notificar de incidencias a los encargados.
    c. **Clientes**:
        i. Visualizar documentación relacionada con su hogar.
        ii. Visualizar imágenes del progreso de la reforma.
        iii. Obtener información general sobre las diferentes obras en caso de tener más de una.
3. Planificar y organizar un proyecto completo desde una etapa temprana.
4. Diseñar un sistema intuitivo y fácil de utilizar para los diferentes perfiles.
5. Diseñar un sistema rápido y escalable (ser capaz de resolver alto tráfico).

1.1.2. Objetivos Secundarios

1. Diseñar un sistema modular, que permita implementar nuevas funciones.
2. Diseñar un sistema adaptable (Responsive), que permita visualizar los componentes en diferentes dispositivos de diferentes tamaños.
3. Aprender a utilizar tecnologías y herramientas modernas.

1.2. Motivación

Implementar un sistema informático conlleva retos y obstáculos, los cuales habrá que aprender a gestionar y resolver. Estos retos van des del primer kick off con el cliente donde es necesario interpretar sus necesidades y trasladarlas al campo tecnológico hasta resolver las diferentes incidencias resultantes de la combinación de tecnologías y la puesta en producción o despliegue del producto.

Aceptar desarrollar este proyecto, me ha ayudado a poner en práctica los conocimientos adquiridos durante la carrera y acabar de consolidarlos. Gracias a asignaturas como Sistemas de comercio electrónico, Técnicas avanzadas de programación, Programación y proyectos de sistemas informáticos, Interacción Persona-Ordenador y Análisis y Diseños de aplicaciones, entre otras, me ha resultado más ameno la gestión, diseño e implementación de este proyecto. Además, me han otorgado una base tecnológica necesaria que ha reforzado la seguridad y confianza para aceptar y continuar con este reto.

Asimismo, ayuda a introducirte en el mercado laboral aprendiendo la dinámica empleada por las empresas del sector para gestionar diferentes proyectos.

Finalmente, se trata de una gran oportunidad para adquirir nuevas habilidades tecnológicas, reforzar los conocimientos adquiridos y aprender tecnológicas de vanguardia. Sin embargo, además de lo anterior, la gestión de este proyecto ayuda también a mejorar las habilidades trasversales (o también conocidas soft skills), mediante la comunicación, discusión y debate con el cliente.

1.3. Planificación

El proyecto se inició la primera semana de octubre del año 2020 y ha tenido una duración de un año aproximadamente.
A continuación, detallaremos la planificación que se ha llevado a cabo:
Planificación y análisis: En el primer mes nos hemos centrado en definir el proyecto, los objetivos principales, secundarios, y los requisitos del cliente.
Diseño: En la fase de diseño, hemos empezado a trazar una solución que engloba todos los objetivos detallados anteriormente.
Implementación: En este apartado, hemos justificado la elección de las tecnologías empleadas e indicado el periodo de desarrollo de la lógica del proyecto.
Integración: Una vez desarrollada la lógica de negocio necesaria, empezamos a unir los diferentes componentes que forman parte del sistema, así como, la conexión entre el front-end y el back-end.
Pruebas: En las pruebas, hemos comprobado el correcto funcionamiento de cada componente del sistema, de este modo, podemos prevenir futuros errores.
Despliegue: Finalmente, preparamos el proceso de despliegue del proyecto hacia los servidores de producción.

Es importante destacar que hemos utilizado la metodología de desarrollo secuencial (waterfall [2]), en la cual no empezamos una nueva fase hasta haber acabado la anterior. De esta forma, evitamos tener que retroceder, por lo tanto retrasar el ciclo del desarrollo.

Finalmente, cabe mencionar que restan fases por describir, como es la formación del personal o la fase de mantenimiento.

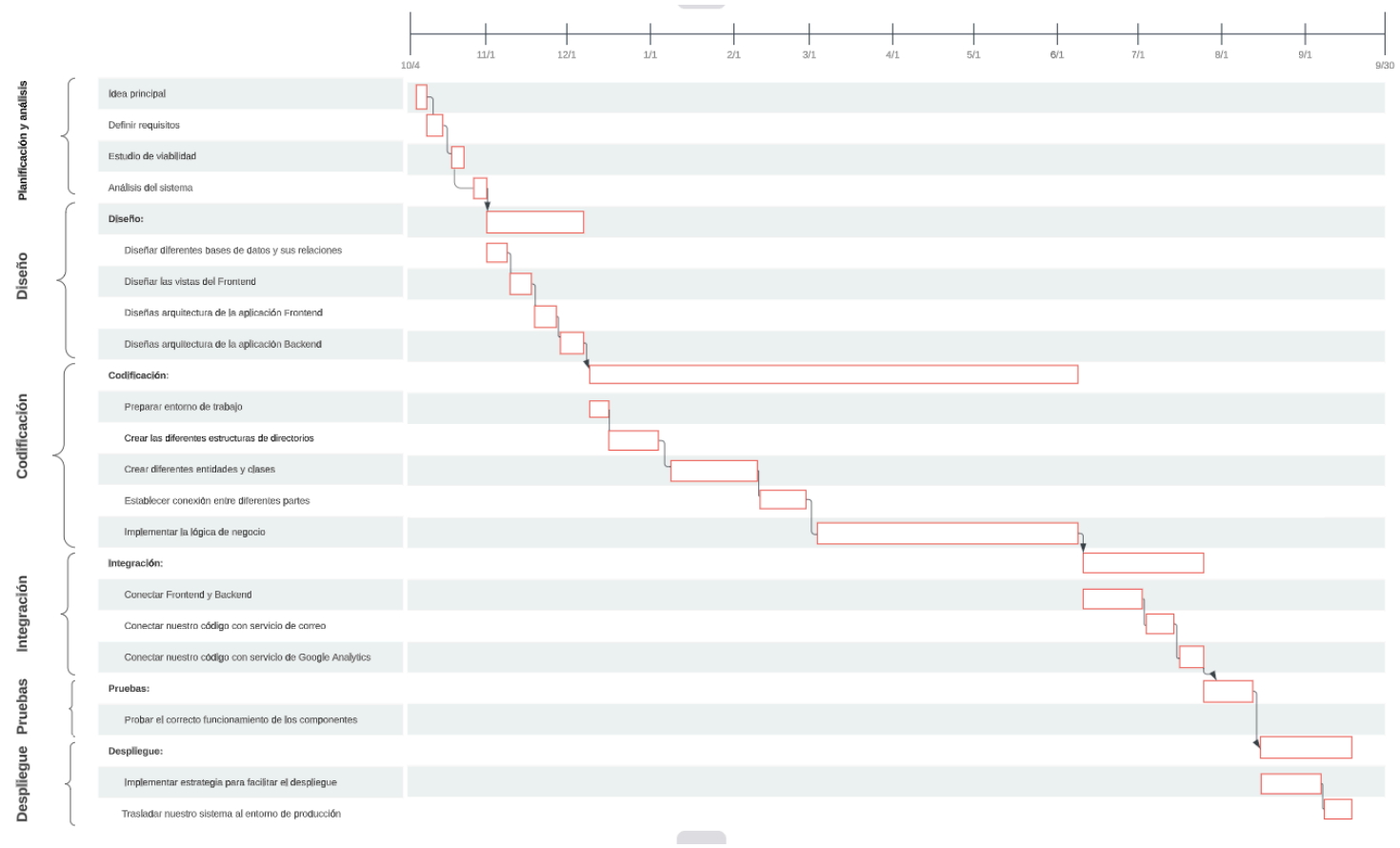


Figura 3. *Diagrama de Gantt. Elaboración Propia.*

1.4. Organización de la memoria

Background:

En este apartado, se definen algunos conceptos y herramientas que consideramos importantes a tener en cuenta antes de empezar con la implementación del sistema. También se presenta el uso de los *framework* en España y más.

Diseño de la propuesta:

En esta sección, hemos fijado tanto los requisitos funcionales como no funcionales que nuestro cliente ha marcado. Además, mostramos los diagramas UML diseñados para ilustrar la solución pensada.

Implementación:

En implementación, vemos como se refleja en la práctica la solución presentada en el apartado anterior. Dicha solución incluye la implementación de dos front-end y dos back-end: para la aplicación administrativa y el sistema de presupuestación.

Evaluación

A la hora de evaluar el proyecto, nos hemos asegurado de cumplir los requisitos propuestos al inicio.

Despliegue

En el despliegue vemos el proceso completo de poner en producción nuestra aplicación utilizando los servicios proporcionados con la plataforma de AWS. El principal objetivo es maximizar el rendimiento y beneficios.

Conclusiones

En última instancia, comentamos como ha sido el proceso, los resultados obtenidos, y no menos importante, la experiencia y conocimientos adquiridos.

2. Background

Empezamos la memoria con unas definiciones sobre herramientas, tecnologías y conceptos necesarios para el desarrollo de este proyecto. Esta sección está organizada de la siguiente forma: i) *Introducción al desarrollo web*; ii) *Bases de datos*; iii) *Experiencia de usuario [10]*.

2.1. Introducción al desarrollo web

El desarrollo web [1] hace referencia al proceso de crear, construir y mantener sitios y aplicaciones web. Este desarrollo engloba una serie de tareas y habilidades de diferentes ámbitos, como es la programación, diseño de interfaces de usuario, administración de las bases de datos y otros aspectos tecnológicos para lograr un correcto funcionamiento.

Años atrás, la experiencia web de un usuario era limitada debido a que los sitios web apenas requerían de la interacción del usuario. En cambio, en los últimos años, el desarrollo web se ha utilizado para crear diferentes aplicaciones de diferentes campos, donde cada una de ellas, exige a los desarrolladores disponer de muchos conocimientos de varias tecnologías y herramientas, lo que les dificulta centrarse en unas herramientas en concreto. Para ello, se ha creado una separación lógica, donde cada una toca diferentes aspectos:

- ***Front-end:*** Los programadores de front-end se centran en la parte visual y la interactividad de un sitio web.
- ***Back-end:*** En cambio, los desarrolladores de back-end se encargan de toda la parte de datos, es decir, crear y mantener estructuras de datos eficientes, procesamiento de datos y la comunicación con las bases de datos y otros sistemas

Cabe destacar que actualmente nos podemos encontrar con programadores Full-stack, que se encargan de ambas partes, del front-end y del back-end.

El desarrollo web abarca diferentes áreas y tecnologías, acto continuo comentaremos estas áreas:

2.1.1. Lenguajes de Programación Web

Un lenguaje de programación [3] son sistemas formales que se utilizan para escribir instrucciones y algoritmos, es decir, es la manera de hacerle entender a una computadora que queremos conseguir.
Los lenguajes de programación más utilizados en el desarrollo web son HTML, CSS, JavaScript.

2.1.2. *Frameworks y bibliotecas*

Los frameworks [45] son una estructura de software que nos proporcionan una base para desarrollar nuestra aplicación. Un framework está formado por un conjunto de herramientas, bibliotecas, componentes y patrones de diseño que vienen a facilitar tanto la creación, implementación como el mantenimiento de la aplicación.
Se trata de agilizar el trabajo a los desarrolladores proporcionándoles componentes y funcionalidad comunes y esenciales en cada proyecto, permitiéndoles enfocarse en la lógica específica de su aplicación.

Los framework comúnmente utilizados son:
- *Front-end*: React, Angular, Vue.js.
- *Back-end:* Django, Ruby on Rails o Node.js.

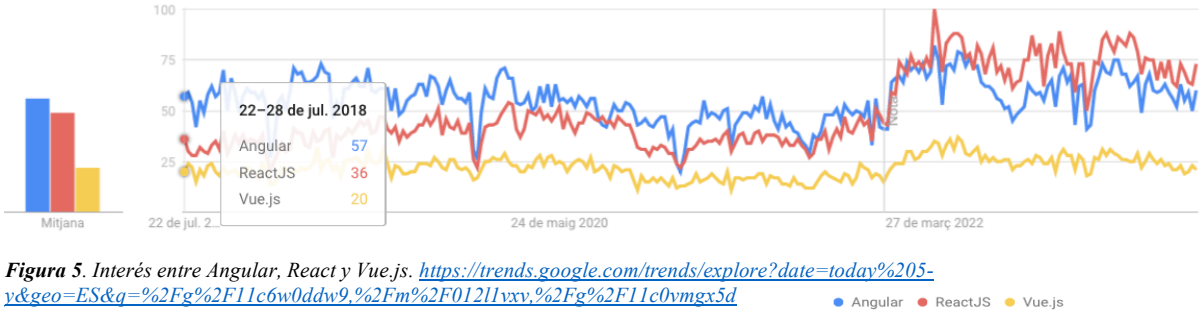


Figura 5. *Interés entre Angular, React y Vue.js.* https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fg%2F11c6w0ddw9,%2Fm%2F012l1vxv,%2Fg%2F11c0vmgx5d

La Figura 5 muestra la evolución del interés de los tres frameworks en España durante los últimos cinco años, como podemos notar, Angular siempre ha tenido interés y va desde 30% hasta 82%, mientras que React tiene una tendencia alcista, empezando con un valor de 36% hasta 73% pasando por un pico de 100% el 02-04-2022. Finalmente, se puede decir que Vue.js tiene un interés bajo, ya que oscila entre 11% y 35%.

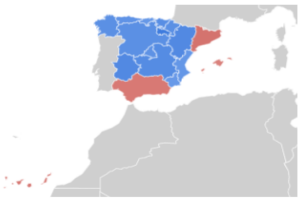


Figura 4. *Desglose comparado por subregión:* https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fg%2F11c6w0ddw9,%2Fm%2F012l1vxv,%2Fg%2F11c0vmgx5d

En la Figura 4, se observa claramente que, en el territorio español, los frameworks más populares son Angular y React. En general, Angular es ampliamente utilizado en toda España, excepto en Cataluña, Andalucía y Canarias, donde se prefiere la utilización de React.

En cambio, el framework de back-end más utilizado en los últimos cinco años en España es sin duda Node.js (ver Figura 6), ya que se basa en JavaScript, permite la gestión simultánea de peticiones, facilita la creación de aplicaciones de gran competencia y, sobre todo, tiene una gran comunidad. [41]

Figura 6. *Interés entre Django, Ruby on Rails y node.js:* https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fm%2F06ff5,%2Fm%2F06y_qx,%2Fm%2F0bbxf89

2.2. *Bases de datos*

Las bases de datos [4] son sistemas organizados y estructuras que se utilizan para almacenar y recuperar volúmenes de datos de manera eficiente. En caso de una aplicación web, se utilizan las bases de datos para almacenar recursos.

En el marcado nos podemos encontrar con una gran variedad de bases de datos, a continuación, explicaremos cada tipo.

2.2.1. *Bases de datos relacionales*

Las bases de datos relacionales (RDBMS), como su nombre indica, se utilizan cuando tenemos datos correlacionados, a modo de ejemplo, una aplicación que requiere un soporte transaccional y una integridad referencial.

El término transaccional [5] hace referencia a la capacidad de garantizar que un conjunto de operaciones se realice de manera segura, consistente y manteniendo la integridad de los datos. Una transacción debe cumplir las propiedades ACID ver Figura 7.

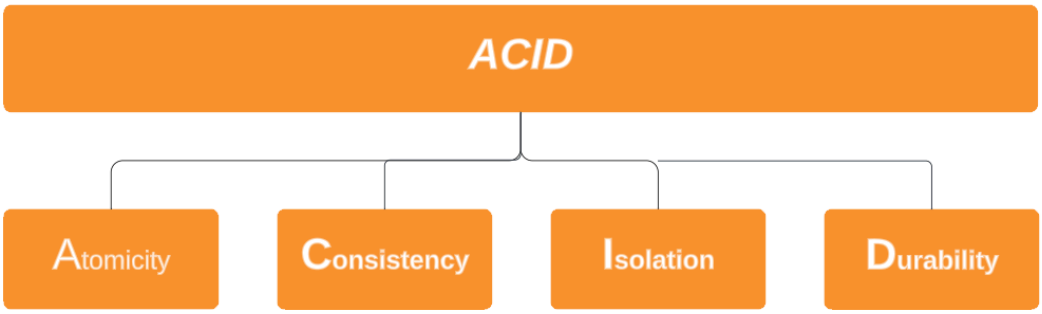


Figura 7. Diagrama ACID. Elaboración Propia.

a) **Atomicidad (Atomicitiy)**: Todas las operaciones en una transacción se realizan como una sola unidad o entidad, es decir, los cambios sobre nuestros datos, o se realizan todos o ninguno, por ejemplo, una aplicación bancaria, donde los usuarios transfieren fondos entre sí, debemos evitar quitar dinero en una cuenta y no transferirlo a la otra.

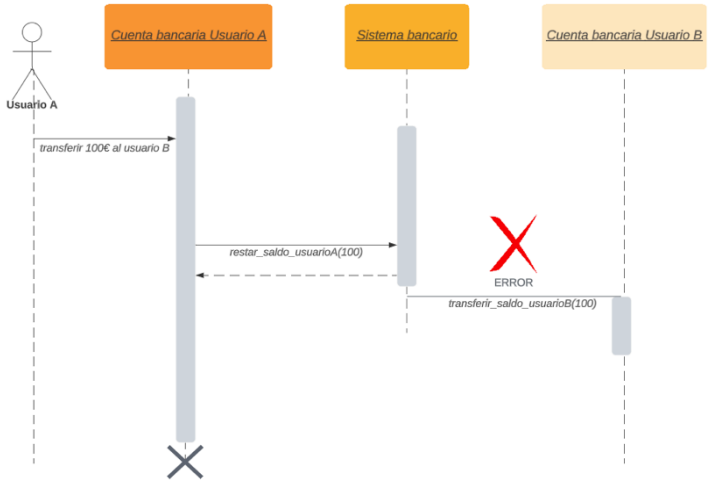


Figura 8. *Atomicitiy – Ejemplo operación bancaria. Elaboración Propia.*

b) **Consistencia (Consistency)**: La consistencia de datos hace referencia a que después de realizar un cambio en la base de datos, este tiene que mantener una coherencia. Por ejemplo, como vemos en la Figura 9, el usuario A tiene en su saldo 2000€ y el usuario B tiene 1800€. Cuando, el usuario A transfiere 200€ al usuario B, su saldo debería actualizarse a 1800€ mientras que el saldo del usuario B se actualizará a 2000€.

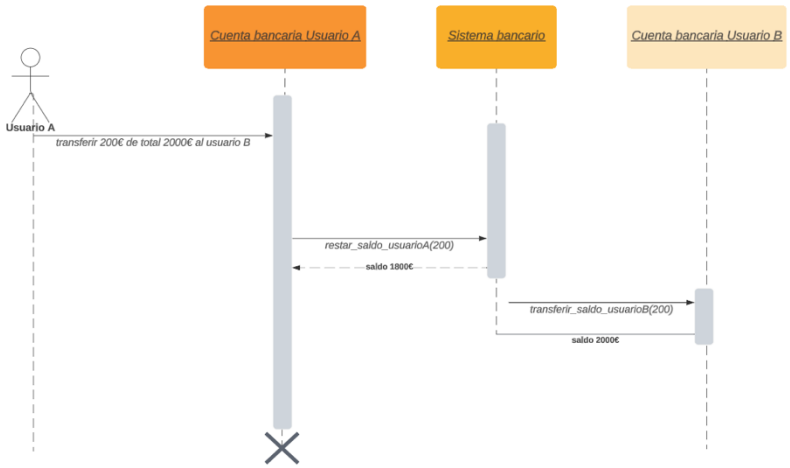


Figura 9. *Consistency – Ejemplo operación bancaria. Elaboración Propia.*

c) ***Aislamiento*** *(Isolation):* Cada transacción se ejecuta de manera aislada respecto a las demás, es decir, al ejecutar una transacción, el resultado de esta no será visible a las demás hasta que acabe la primera.

d) ***Durabilidad*** *(Durability):* Esta última propiedad nos indica que una transacción ejecutada con éxito, su resultado no se podrá deshacer, es decir, será permanente, incluso en caso de fallos en el sistema.

2.2.1.1. Ejemplos de Bases de datos relacionales

Finalmente, proporcionaremos un listado con los nombres de las bases de datos relacionales comúnmente utilizadas [6]:

1. Oracle Database
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL
5. SQLite

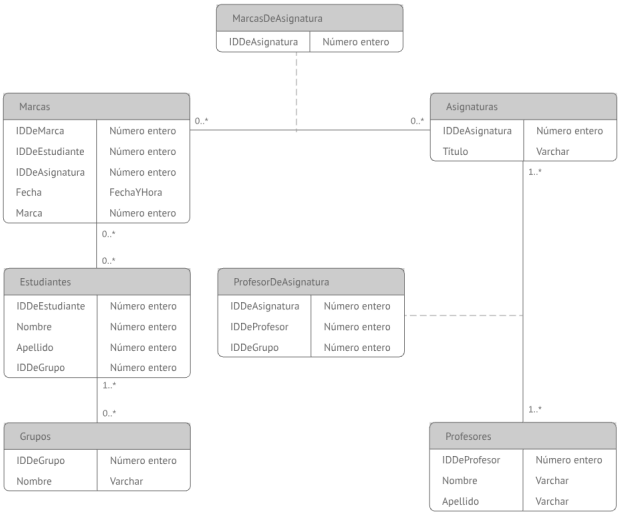


Figura 10*. Diagrama relacional:*
https://www.lucidchart.com/pages/es/ejemplos/herramienta-ERD

2.2.2. *Bases de datos NoSQL*

Las bases de datos NoSQL [8] (Not Only SQL) difieren de los sistemas relacionales en su diseño y enfoque. Las bases de datos NoSQL sirven para almacenar datos no estructurales, semi-estructurados o horizontales.

Las NoSQL son convenientes para las aplicaciones con un volumen de datos alto, como, por ejemplo, análisis de Big data o aplicaciones web en tiempo real.

2.2.2.1. *Características*

Las bases de datos no relacionales constan de las siguientes características [7][9]:

- **Datos flexibles**: Los datos se pueden almacenar de diferentes formas, tanto documento, grafos, par clave-valor o columnas. Esta flexibilidad a la hora de almacenar los datos nos permite recuperarlos de manera rápida y sin un esquema fijo.
- **Escalabilidad horizontal**: Las bases de datos NoSQL están diseñadas para escalar de manera horizontal, es decir, en vez de tener los datos en un único servidor grande, los almacenamos en diferentes servidores distribuidos, lo cual permite un mayor rendimiento y capacidad de almacenamiento a medida que el volumen de datos aumentan.
- **Alta disponibilidad y tolerancia a fallos**: Al distribuir los datos en diferentes servidores, si uno falla, podemos solicitar los datos a otro servidor, ya que la información está replicada en cada uno de ellos.
- **Operaciones de alta velocidad**: Los sistemas NoSQL están optimizados para realizar operaciones de lectura y escritura de alta velocidad, este hecho es adecuado para aquellas aplicaciones que requieren de un rendimiento rápido y de baja latencia.
- **Lenguaje de consultas alternativos**: Finalmente, en lugar de usar SQL para realizar consultas, podemos emplear un lenguaje alternativo o interfaces de usuario para interactuar con los datos.

2.2.2.2. *Ejemplos Bases de datos NoSQL*

Bases de datos no relacionales comúnmente utilizadas [7]:

1. MongoDB
2. Cassandra
3. Redis
4. CouchDB
5. Neo4j

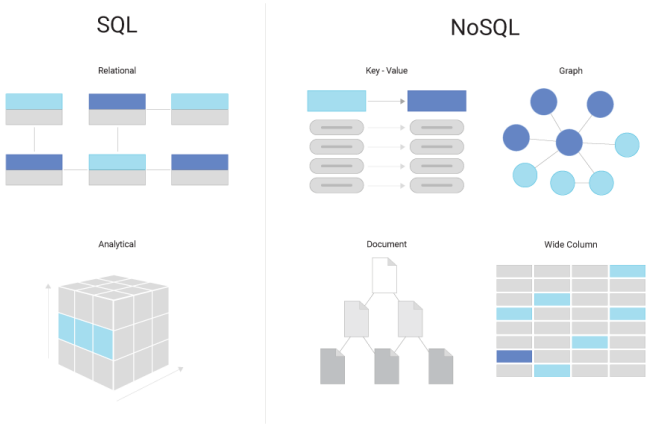


Figura 11.
https://www.scylladb.com/glossary/nosql-design-principles/

2.2.3. Otras bases de datos

A parte de las bases de datos relacionales o no relacionales, existen otros tipos de BD, no tan comúnmente utilizadas en el desarrollo web.

2.2.3.1. Bases de datos de objetos

Este tipo de bases de datos permiten almacenar y recuperar objetos complejos de forma eficiente.
Son convenientes para aquellas aplicaciones que tienen un modelo de datos orientado a objetos, como sistemas de gestión de objetos, sistemas CAD y, sobre todo, para sistemas de simulación.
Bases de datos de objetos comúnmente utilizadas*:* db4o, ObjectDB y Versant.

2.2.3.2. Bases de datos en memoria

Podemos afirmar que este tipo de BD son para los datos delicados, es decir, cuando es necesario un acceso extremadamente rápido y de alto rendimiento.
Un ejemplo de uso común son aquellas aplicaciones que requieren una respuesta en tiempo real, como un sistema de análisis de datos en tiempo real o un sistema de videojuegos en línea.
Bases de datos en memoria comúnmente utilizadas: Redis, Memcached, Apache Ignite y VoltDB.

2.2.3.3. Bases de datos especiales

las BD especiales son utilizadas cuando es necesario guardar o consulta datos relacionados con las ubicaciones geográficas o formas geométricas.
El caso de uso habitual son los sistemas de información geográficos (SIG), navegación o logística.

Bases de datos especiales comúnmente utilizadas: PostGIS (extensión espacial de PostgreSQL), MySQL Spatial (extensión espacial de MySQL), Oracle Spatial and Graph, MongoDB con soporte espacial (mediante índices geoespaciales).

2.2.3.4. Bases de datos temporales

las BD temporales son necesarias en aquellos casos que requieren de un rastreo o consultar cambios en los datos a lo largo del tiempo.
Son utilizadas en el ámbito de auditorías, sistemas de historial de versiones y aquellas aplicaciones que deben tener un seguimiento detallado de los cambios.

Bases de datos temporales comúnmente utilizadas: Oracle Workspace Manager, PostgreSQL con extensión temporal tables.

2.3. Experiencia de usuario [10]

El diseño web se centra en la creación y el diseño de la apariencia visual de la aplicación web. El cual tiene como objetivo principal lograr una interfaz de usuario atractiva y funcional que brinde una experiencia de usuario positiva y amena. Para ello se debe planificar, crear, estructurar y diseñar las interfaces de manera correcta.

Los desarrolladores a la hora de diseñar un sistema web, es clave que anteriormente hayan implementado un prototipo en el cual se ilustran aspectos como la accesibilidad, usabilidad, memorabilidad, la navegación intuitiva y finalmente la optimización para dispositivos móviles.

- **Accesibilidad**: Es la capacidad de una aplicación para ser utilizada y entendida por todas las personas, incluyendo aquellas con discapacidad o limitaciones.
- **Usabilidad**: se refiere a la facilidad con la que los usuarios pueden utilizar una aplicación web y realizar tareas deseadas, dicho de otra forma, facilitar a los usuarios encontrar la información de manera eficiente y rápida.
- **Memorabilidad**, Es la capacidad de una aplicación de hacer que los usuarios puedan recordar las acciones en un futuro.

Las acciones comentadas se pueden agrupar en dos grupos, UI (User Interface) o UX (User Experience).

3. *Diseño de la propuesta:*

En este apartado definimos el diseño y la arquitectura del prototipo a implementar.
Esta sección está formada por los siguientes apartados: i) *Requisitos [11]*; ii) *Diagramas UML*.

Es importante proporcionar una idea general del proyecto previo antes de comenzar a detallar los diferentes requisitos. A continuación, en la Figura 13 ver como se estructura el sistema:

- **Aplicación inicio**: En la aplicación de inicio, el usuario podrá escoger la acción a realizar, ir al sistema de presupuestación o la aplicación administrativa.
- **Sistema de presupuestación**:
    - *Sistema antiguo de presupuestación*: En Figura 12 se puede observar las etapas por las cuales pasa un usuario antes de obtener el coste de su reforma.
      Esta manera de trabajar supone un coste adicional a la empresa, dado que debe mover técnicos o arquitectos para medir la superficie a reformar. Este coste se puede mitigar facilitando un presupuesto al usuario previo al contacto telefónico, de esta forma aumentamos la probabilidad de contratación.

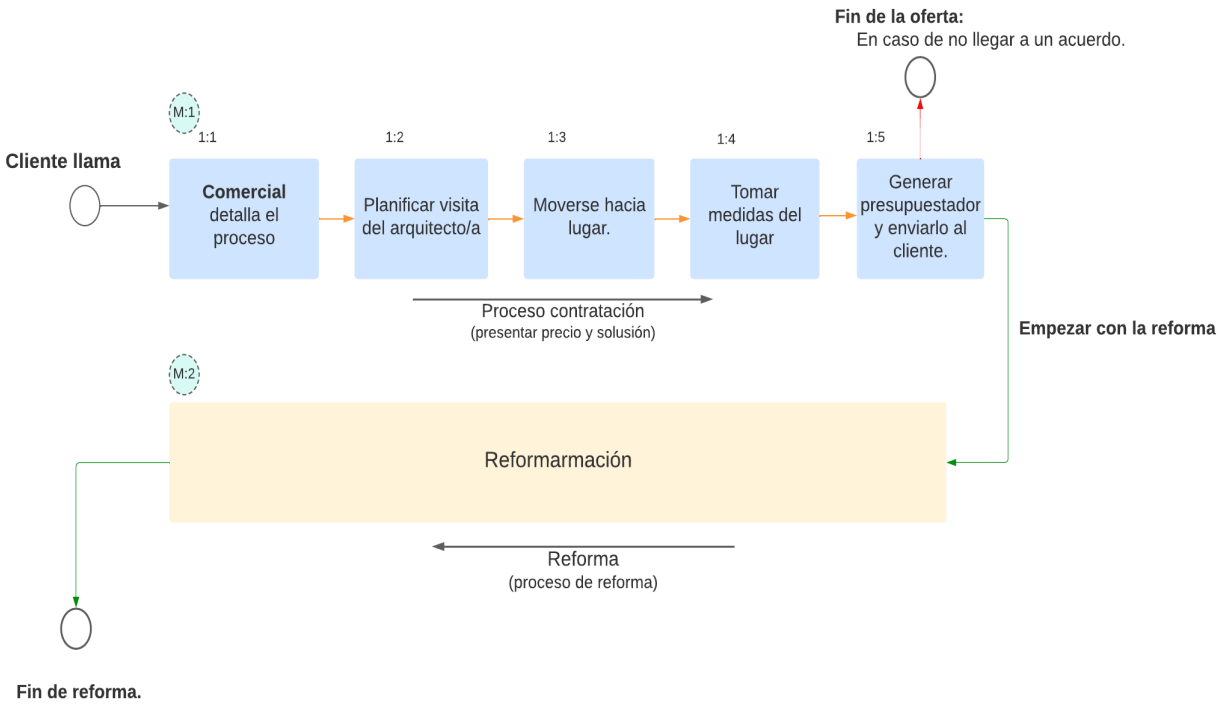


***Figura 12**. Ciclo de la reforma. Elaboración Propia.*

- *Sistema a de presupuestación a implementar:* La idea detrás del sistema de presupuestación es brindar a los usuarios, de manera rápida y fácil, un sistema capaz de calcular el coste de la reforma de su hogar en base a las medidas introducidas. Sobre todo, debe ser preciso y exacto.

- **Aplicación administrativa**: La aplicación administrativa será utilizada por tres roles, donde cada uno tendrá una funcionalidad diferente al resto:
    - *Administradores:* Los administradores podrán realizar todas aquellas tareas de dar de alta a recursos, permitiendo gestionarlos de manera eficiente e intuitiva.

    - *Clientes:* Los clientes tendrán acceso a aquella documentación relacionada con su obra (planos y documentos) y, sobre todo, podrán ver la evolución de su reforma mediante imágenes.

    - *Trabajadores:* Los trabajadores podrán ver las tareas que se les asignen y al mismo tiempo, se les permitirá informar de cualquier incidencia.

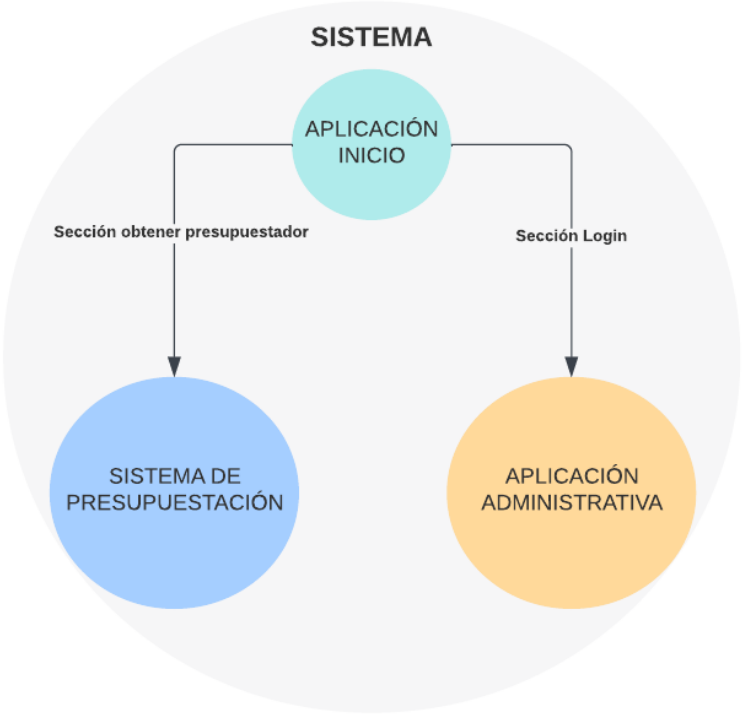


Figura 13. *Diseño del sistema a implementar. Elaboración Propia.*

3.1. *Requisitos [11]*

Los requisitos de nuestro proyecto se han establecido mediante las sesiones iniciales como los comentarios (*feedback*) que hemos ido recibiendo a medida que el proyecto ha ido avanzando. Los requisitos a cumplir son los siguientes:

3.1.1. *Requisitos Funcionales*

Los requisitos funcionales detallan las tareas específicas que nuestra aplicación debe llevar a cabo. En la siguiente línea, enumeraremos estas funciones:

***Tabla 1**. Tareas del sistema de presupuestación. Elaboración Propia.*

Índice	Tarea	Descripción
1	Seleccionar las partes a reformar.	El usuario podrá escoger aquellas partes de la vivienda que desea reformar.
2	Introducir las medidas de aquellas partes seleccionadas.	El usuario introducirá las medidas de las partes seleccionadas.
3	Generar el coste de cada parte.	El presupuestador mediante unos precios y formulas, calculará el coste.
4	Mostrar el coste total de la reforma.	Después de introducir las medidas, el usuario podrá ver el precio total de la reforma, inclusive el precio de cada parte.
5	Permitir al usuario recibir un documento PDF detallado.	Si el usuario desea obtener un documento PDF con más información, podrá solicitarlo introduciendo sus datos mediante un formulario.
6	Generar el presupuesto en forma de PDF.	El presupuestador tiene que ser capaz de generar un PDF especifico con los precios de aquellas partes seleccionadas por el usuario, y, sobre todo, tiene que ser agradable a la vista.
7	Enviar el presupuesto al correo del usuario.	El documento se tiene que enviar a la cuenta de correo electrónico del usuario solicitante.
8	Guardar un registro del presupuesto dado.	Una vez enviado el documento al usuario, el sistema tiene que guardar en la base de datos de la empresa el coste final de la reforma entregado como las medidas que lo componen.

En la tabla anterior, vemos que aparte de calcular el coste que supondrá la reforma, el sistema de presupuestación tiene que ser capaz de registrar toda la información introducida por el usuario. También tiene que generar el documento PDF y adjuntarlo a la cuenta de correo electrónico que el mismo usuario ha facilitado mediante un formulario.

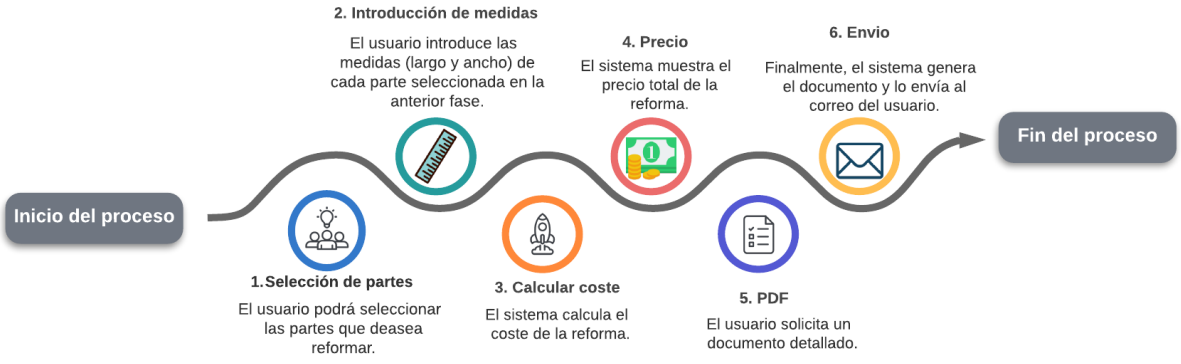


***Figura 14**. Proceso para generar el presupuesto. Elaboración Propia*

***Tabla 2**. Tareas de la aplicación administrativa rol Administrador. Elaboración Propia.*

Índice	Tarea	Descripción
1	Log in.	El administrador podrá iniciar sesión con su cuenta mediante usuario y contraseña.
2	Alta cliente.	Podrá realizar las operaciones CRUD (crear, listar, modificar y eliminar) del cliente.
3	Alta trabajador.	Podrá realizar las operaciones CRUD del trabajador.
4	Alta obra.	Podrá realizar las operaciones CRUD de la obra.
5	Subir foto.	El administrador podrá subir imágenes al sistema y asignarlas a una obra.
6	Publicar foto.	De la lista de imágenes de una obra, el administrador podrá mostrar u ocultar una imagen al cliente.
7	Crear evento.	Podrá crear eventos con una determinada duración.
8	Asignar evento al trabajador	Finalmente, podrá asignar un evento a un trabajador.

***Tabla** 3. Tareas de la aplicación administrativa rol Cliente. Elaboración Propia.*

Índice	Tarea	Descripción
1	Log in.	El cliente podrá iniciar sesión con su cuenta.
2	Ver imágenes obra	El cliente podrá ver la evolución de su obra a través de imágenes.
3	Ver documentos.	Podrá listar los documentos proporcionados por la empresa inmobiliaria.
4	Ver planos.	Podrá ver los planos creados para su obra.

***Tabla 4**. Tareas de la aplicación administrativa rol Trabajador. Elaboración Propia.*

Índice	Tarea	Descripción
1	Log in.	El trabajador podrá iniciar sesión con su cuenta.
2	Listar eventos.	El trabajador podrá ver los eventos que tiene asignados mediante un calendario.
3	Notificar incidencia.	Debido a los desafíos y dificultades que enfrentan los trabajadores en cada reforma, es necesario que puedan informar sobre ellas.

3.1.2. Requisitos No Funcionales

Los requisitos no funcionales describen aquellos aspectos que no están estrictamente relacionados con la funcionalidad de una aplicación, sin embargo, son muy importantes para su rendimiento, seguridad, usabilidad y calidad.

***Tabla 5**. Tareas de la aplicación administrativa rol Trabajador. Elaboración Propia.*

Índice	Tarea	Descripción
1	Vistas intuitivas.	Las vistas deben tener un diseño agradable y amigable tanto para los clientes como para los empleados que usarán la aplicación diariamente.
2	Responsive.	Las vistas se tienen que adaptar a diferentes pantallas de diferentes tamaños.
3	Navegación intuitiva.	Para realizar una acción, esta tiene que resultarle fácil de encontrar al usuario, es decir, el usuario debe llegar a dicha acción en pocos pasos.
4	Capacidad de carga y eficiencia.	A la hora de renderizar nuestra aplicación, esta debe tener un tiempo de carga rápida, ya que, si demora tanto, generará una molestia a los usuarios.
5	Autenticación.	Un recurso solo debe ser accesible por la persona y el rol correspondiente.
7	Cumplir con GDPR	A la hora de subir nuestra aplicación al servidor de producción, esta tiene que cumplir con la normativa general de protección de datos europea.

En la tabla anterior, podemos ver aquellos aspectos a tener en cuenta a la hora de implementar nuestro sistema además de la funcionalidad obligatoria.

3.2. Diagramas UML

En esta sección vemos los diferentes diagramas UML diseñados para ilustrar la solución pensada y sus tipos:

- Diagramas de comportamiento:
  - Diagrama de caso de uso.
  - Diagrama de secuencia.
- Diagramas estructurales:
  - Diagrama de clases.

3.2.1. Sistema presupuestación

En la Figura 15 vemos mediante un diagrama de caso de uso las acciones que puede realizar el usuario para solicitar el coste de su reforma.

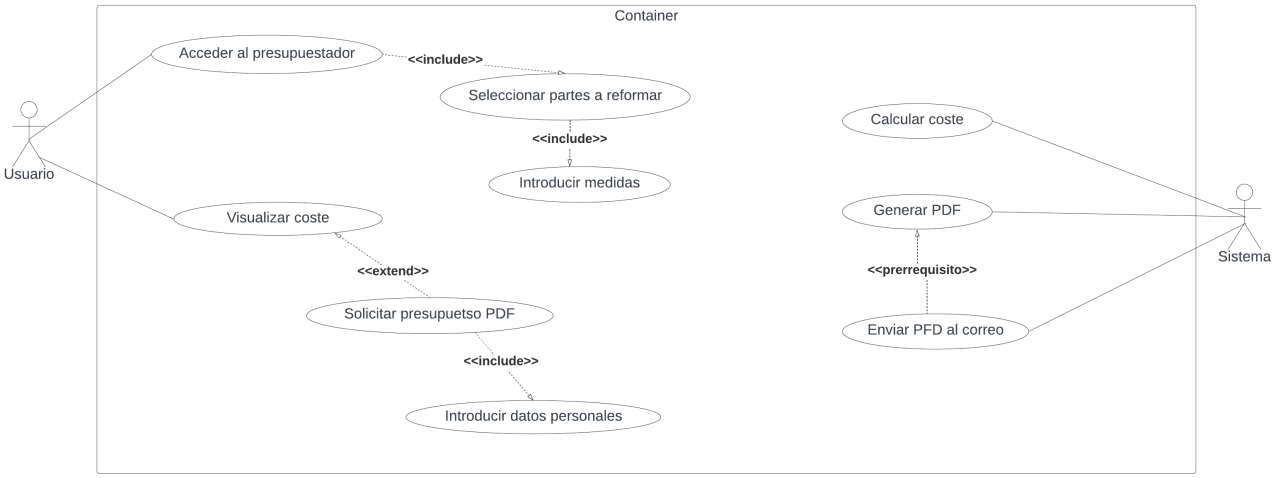


Figura 15. *Diagrama de caso de uso del sistema de presupuestación. Elaboración Propia*

En la Figura 16 se presentan las mismas acciones que en la figura anterior, pero con mayor nivel de detalle mediante el diagrama de secuencia.

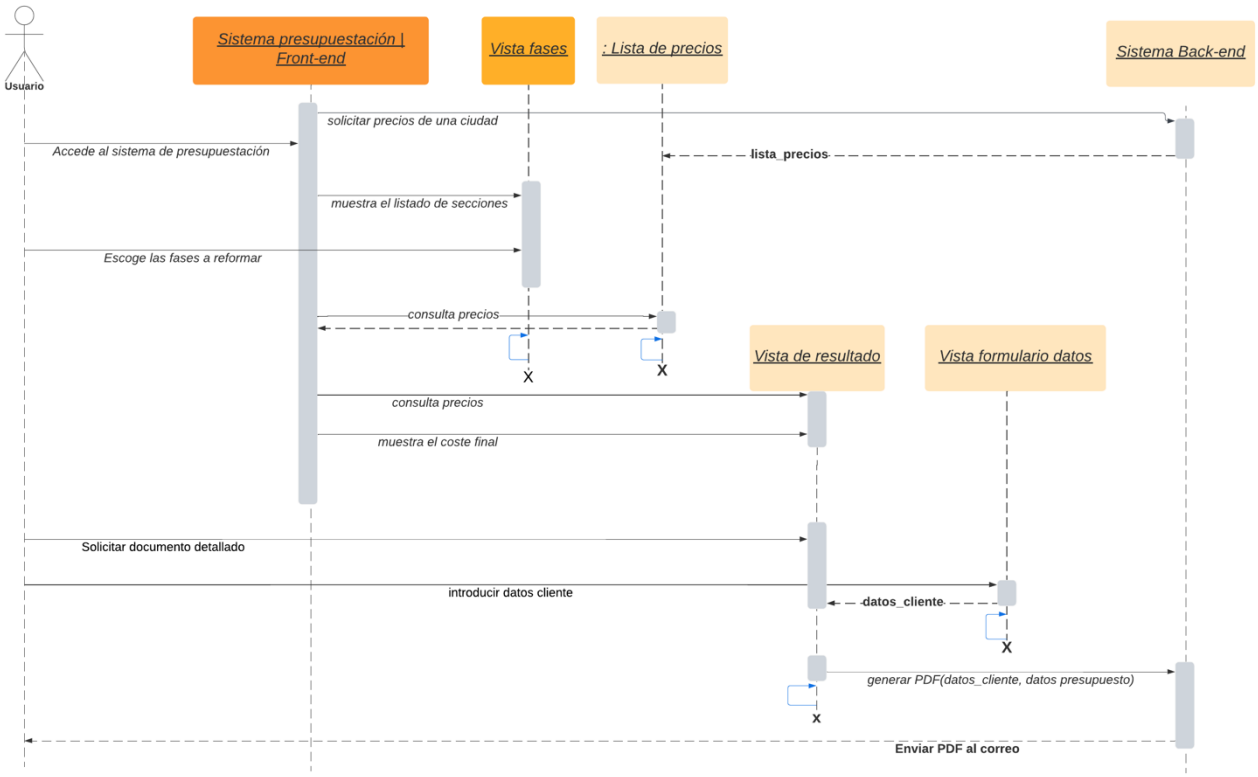


Figura 16. *Diagrama de frecuencia del sistema de presupuestación. Elaboración Propia*

En la figura adjunta, se presenta un diagrama de frecuencia que ofrece una representación visual de las acciones ejecutadas por el usuario, las vistas involucradas y las interacciones del sistema, tanto en el front-end como en el back-end.

1. En primer lugar, el usuario accede al sistema de presupuestación.
2. El sistema muestra una vista para que el usuario pueda elegir las secciones de la casa que desea reformar, cada una de estas dispone de sus medidas, instalaciones, tiempo requerido y sobre todo, personal para llevar a cabo la reforma.
3. Al mismo tiempo que la aplicación permite al usuario personalizar su obra, esta solicita los precios al back-end, de esta forma se anticipa para cuando tenga que realizar los cálculos.
4. El usuario debe escoger e introducir las medidas y otros datos necesarios para un presupuesto exacto.
5. La aplicación basándose en los datos del usuario, calcula el coste que supondrá la reforma.
6. Tras completar los cálculos, el resultado total se le mostrará al usuario en una vista aparte.
7. Después de visualizar el coste, si el usuario desea recibir un documento PDF con más detalles, deberá completar un formulario que incluya su nombre, teléfono y dirección de correo electrónico.
8. El front-end enviará una solicitud HTTP hacia el back-end con los datos del presupuesto y los datos del cliente. Luego, el back-end generará un PDF y enviará el documento a la cuenta de correo del usuario. Finalmente, se registrará el coste proporcionado al usuario junto con la fecha.

3.2.2. *Aplicación administrativa*

En esta sección vemos mediante un diagrama de caso de uso las acciones que puede realizar el administrador dentro del sistema. El diagrama de la aplicación administrativa es bastante extenso, por lo tanto, lo dividiremos en los casos de uso de los diferentes roles.

3.2.2.1. *Rol administrativo*

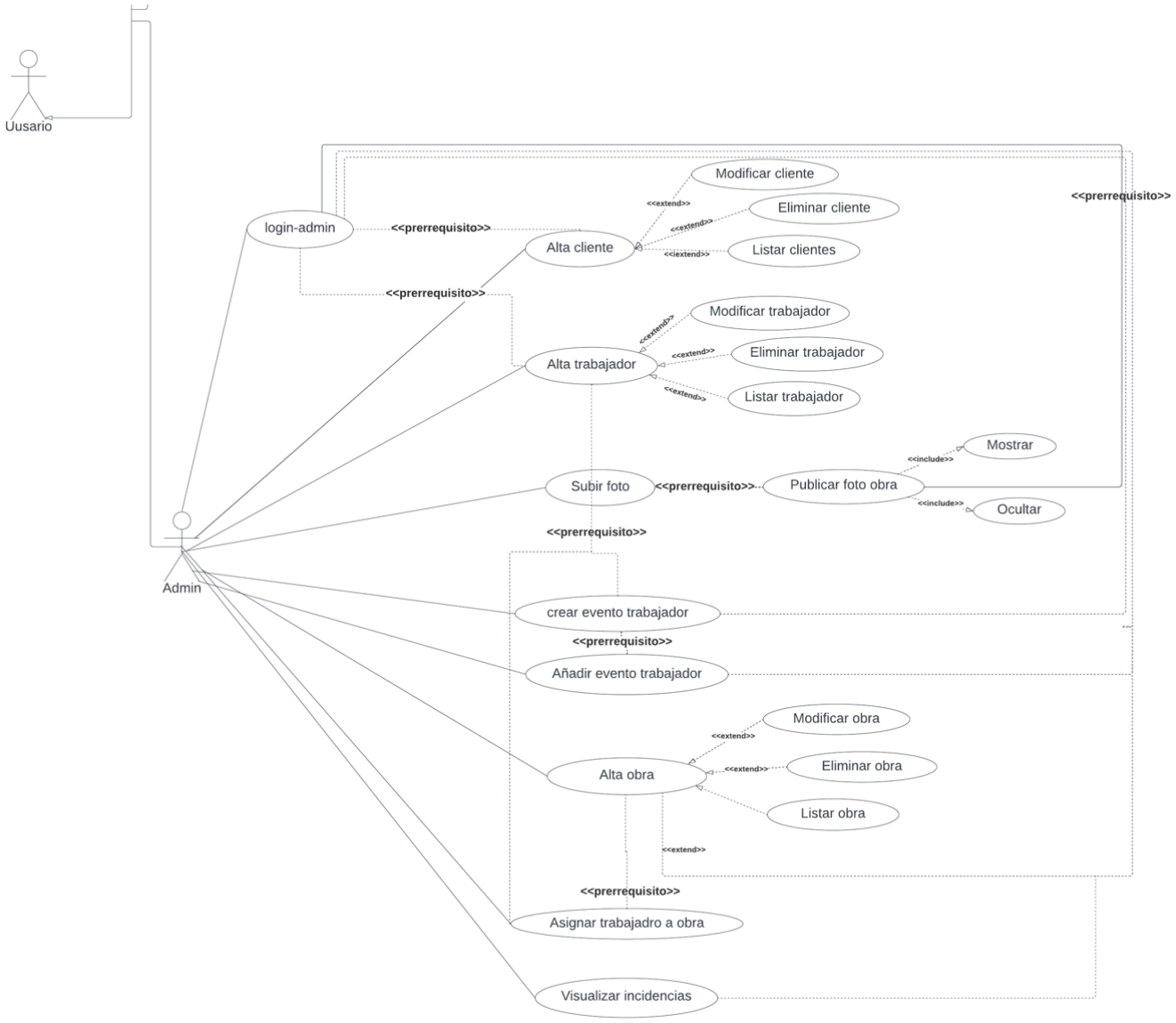


Figura 17*. Diagrama de caso de uso del administrador. Elaboración Propia*

Como podemos observar en la Figura 17, el administrador puede efectuar muchas tareas, pero primero debe iniciar sesión con sus credenciales correspondientes.
Las acciones que puede realizar el administrador son las siguientes:

1. En primer lugar, iniciar sesión con su cuenta.
2. Un cliente después de contratar los servicios de la empresa, este será dado de alta en la aplicación por el administrador. También podrá realizar las operaciones CRUD del cliente.
3. El administrador tendrá disponible todo el CRUD de los trabajadores que forman parte de la plantilla, es decir, podrá listar, modificar o eliminar a un trabajador.
4. Además de subir imágenes de la obra (por defecto no serán visibles al dueño de la obra), el administrador podrá listar las fotos de la obra, filtrarlas por diferentes aspectos y, sobre todo, publicar u ocultar una imagen al cliente.
5. El administrador tendrá acceso a otro conjunto de operaciones CRUD relacionadas con las obras. Al crear una obra, es necesario asignarle un cliente, el cual debe ser añadido en el sistema con antelación.
6. Después de contar con los recursos fundamentales (obra, cliente y trabajadores) a disposición, el administrador estará facultado para asignar trabajadores a obras específicas. Mediante esta acción, se logrará una organización más eficiente de los eventos.
7. Una vez que se hayan asignado los trabajadores, será posible crear y asignar eventos a cada uno de ellos.
8. Finalmente, el administrador podrá listar todas las incidencias que se hayan generado.

3.2.2.2. *Rol cliente*

El propósito central de la empresa es involucrar a los clientes en todo el proceso de remodelación. En otras palabras, se busca evitar situaciones en las que los clientes carezcan de información sobre el progreso de sus obras, desde la entrega de las llaves hasta la finalización de estas. Además de esto, se persigue optimizar y focalizar el envío de los documentos vinculados a cada obra.

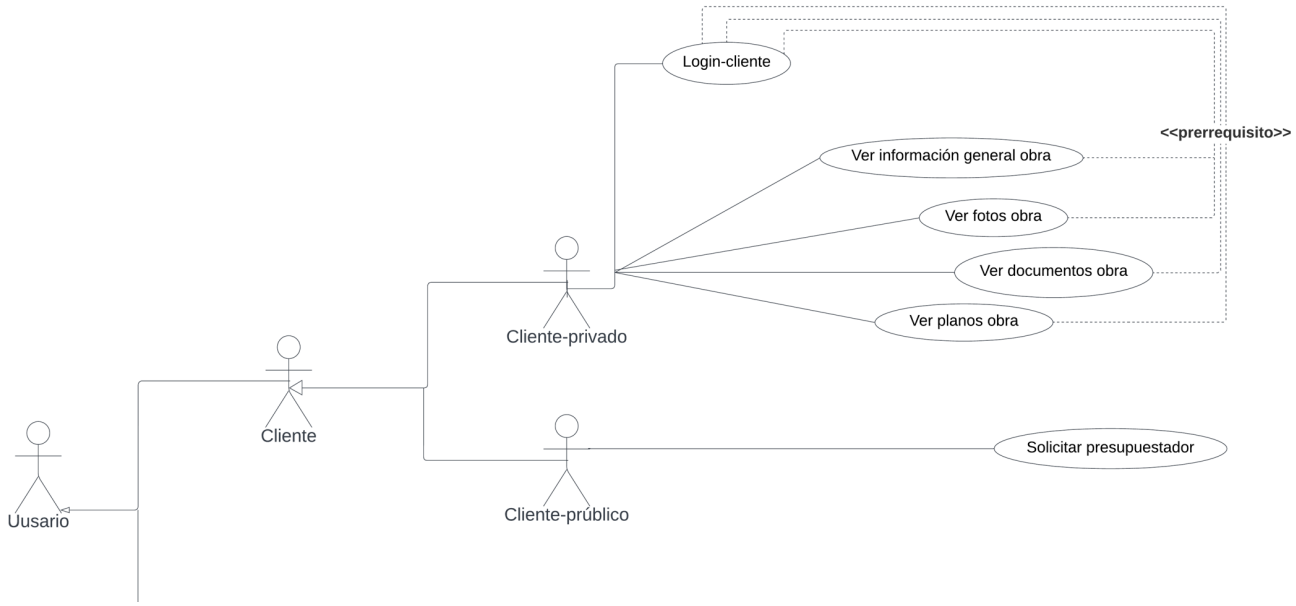


Figura 18. *Diagrama de caso de uso del cliente. Elaboración Propia*

En la Figura 18, es evidente que el rol de usuario se divide en dos segmentos distintos: el cliente público y el cliente privado. El cliente público representa al usuario que accede al sistema de presupuestación con el objetivo de solicitar una estimación del costo para una remodelación. Por otro lado, el cliente privado se refiere al individuo que ya ha formalizado un contrato para llevar a cabo dicha remodelación.

El cliente privado puede realizar las siguientes acciones dentro de la aplicación administrativa:

1. En primer lugar, el cliente deberá iniciar sesión con su cuenta, dado que la aplicación tiene la responsabilidad de asegurar las rutas. Esto implica que un rol específico solo tendrá acceso a las acciones autorizadas para ese rol en particular.
2. Una vez ingresado a la aplicación, el cliente podrá ver aquellos documentos relacionados con su obra.
3. También podrá ver las imágenes de su obra agrupadas por fases, es decir, para permitir al cliente navegar entre un conjunto de fotos de manera fácil, el administrador a medida que va publicando las imágenes las irá clasificando por fases, donde la fase hace referencia a una etapa del ciclo de la reforma.
4. Finalmente, habrá una sección en la aplicación donde ofrecemos al cliente información general como el número de obras que ha contratado, el número de documentos de los que dispone y más información.

3.2.2.3. Rol trabajador

En este apartado vemos los casos de uso que un trabajador puede realizar dentro de la aplicación.

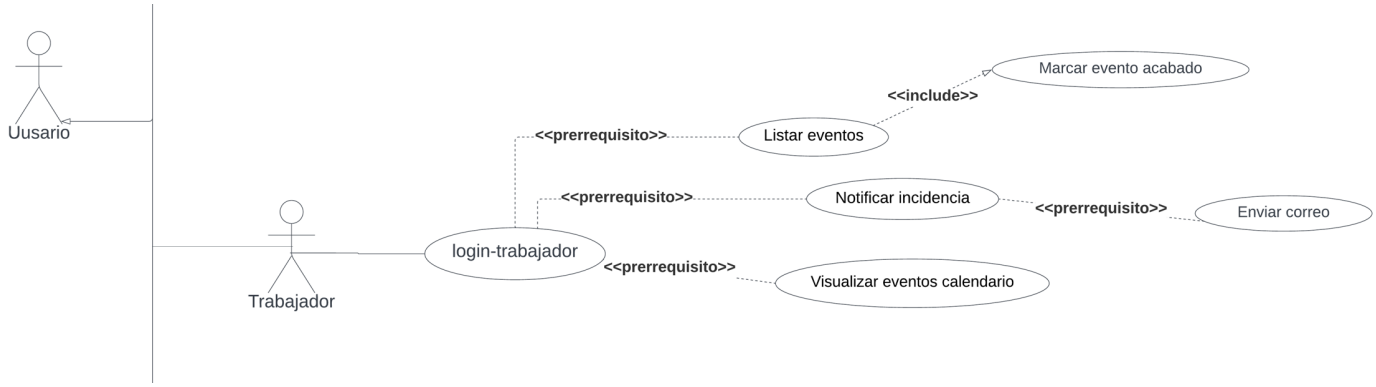


Figura 19. *Diagrama de caso de uso del cliente. Elaboración Propia*

1. Como es de costumbre, la primera acción a realizar es el inicio de sesión como trabajador.
2. Además de visualizar los eventos asignados, el trabajador también tiene la capacidad de marcarlos como completados a medida que los va concluyendo.
3. También tiene la capacidad de listar aquellos eventos que ya han sido finalizados. Esta opción puede resultar útil en momentos en los que sea necesario llevar un conteo o verificar el estado histórico de un evento en particular.
4. Por último, el trabajador tendrá la posibilidad de reportar cualquier incidencia relacionada con una obra. Esta acción generará una notificación automática al administrador a través de correo electrónico.

3.2.3. *Diseño de la base de datos*

En esta sección, presentamos los diagramas relacionados con la estructura de la base de datos implementada para guardar los precios que el sistema de presupuestación necesita para realizar los cálculos correspondientes.

3.2.3.1. *Diseño del sistema de presupuestación*

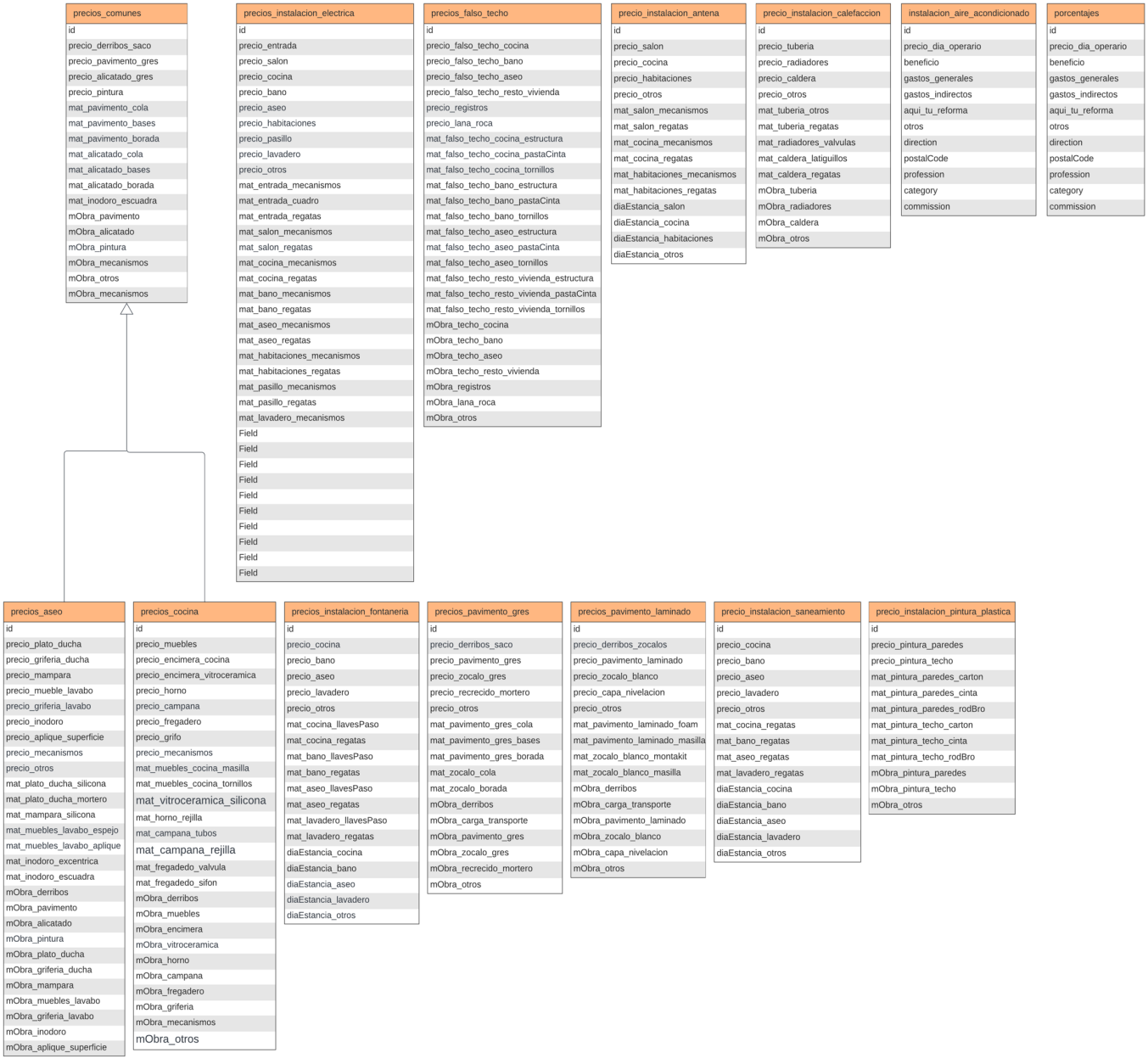


Figura 20. *Diagrama de clases del sistema de presupuestación. Elaboración Propia*

Como podemos observar en la Figura **20**, las tablas de los precios no están relacionadas entre sí, por lo tanto, podemos utilizar una base de datos de tipo NoSQL, de esta forma, recuperaremos los precios de forma rápida y eficaz.

3.2.3.2. *Diseño del sistema de la aplicación administrativa*

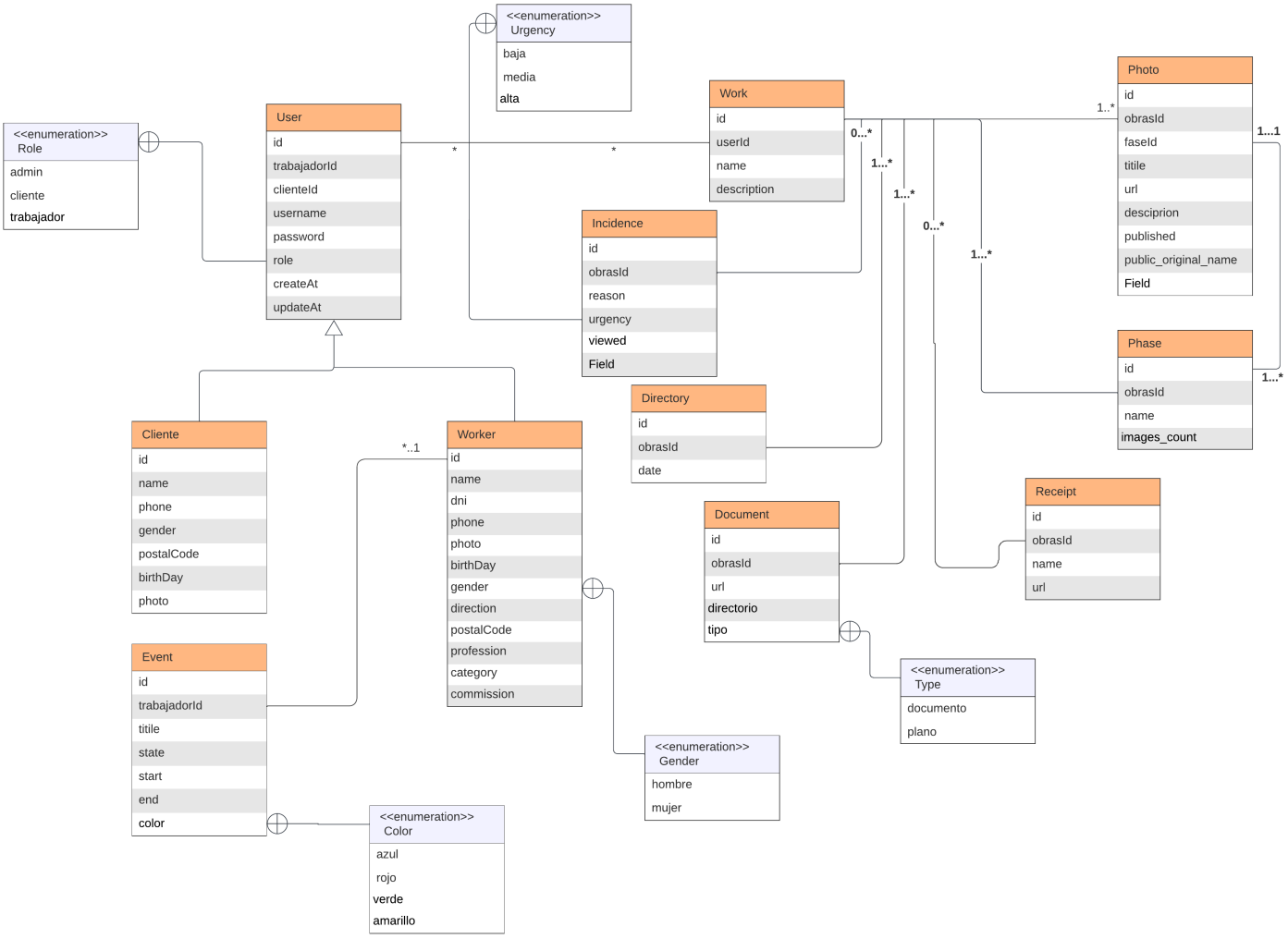


***Figura 21**. Diagrama de clases de la aplicación administrativa. Elaboración Propia*

A continuación, procedemos a describir las clases que componen la Figura 21.
En la parte izquierda de la figura, se presenta la clase *User*. Esta clase contiene información general acerca de un usuario y al mismo tiempo establece una relación de herencia con las clases hijas *Cliente* y *Trabajador*. Esta estructura permite a nuestros usuarios asumir diversos roles en el sistema.
Además, se observa una relación entre la clase *Trabajador* y *Evento*. Esta asociación nos permite almacenar los eventos asignados a un trabajador en específico.
En la parte de la derecha del mismo diagrama, notamos la presencia de la clase *Obra*. Esta clase, además de contener la información propia de una obra, establece diversas relaciones con las demás entidades. Esta estrategia permite guardar el identificador de una obra en las clases relacionadas, lo que facilita la recuperación de la información necesaria con un número reducido de consultas. Este enfoque contribuye a una gestión eficiente de recursos en el largo plazo.
Finalmente, podemos ver la implementación de diferentes enumeraciones (*enums*) para limitar el valor que puede tener un campo. Cabe comentar que, para este tipo de relaciones, utilizamos una base de datos relacional (MySQL).

4. *Implementación [12]*

En esta sección nos centramos en poner en práctica las diferentes alternativas diseñadas y llevadas a cabo para la creación del sistema.

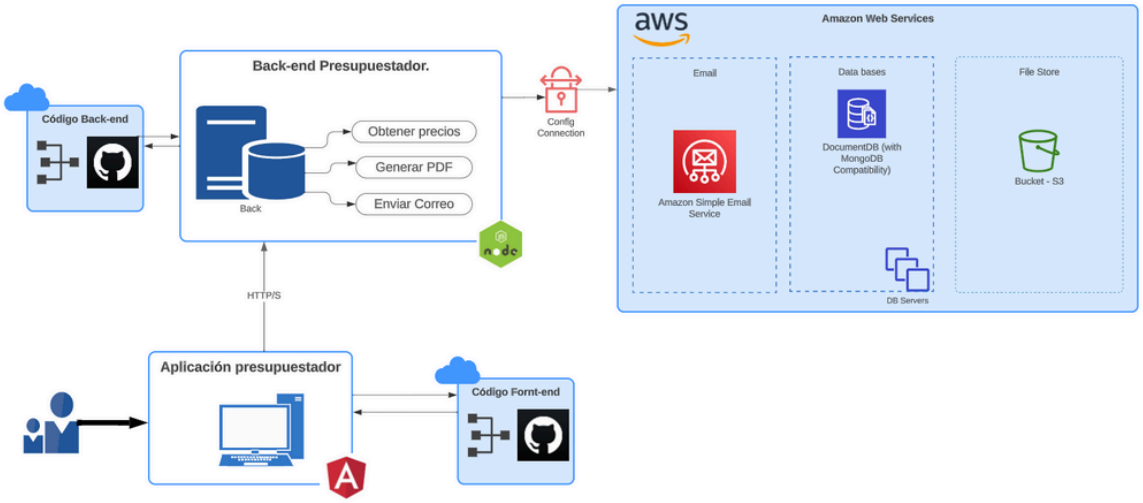


***Figura 23**. Diagrama sistema de presupuestación final.*

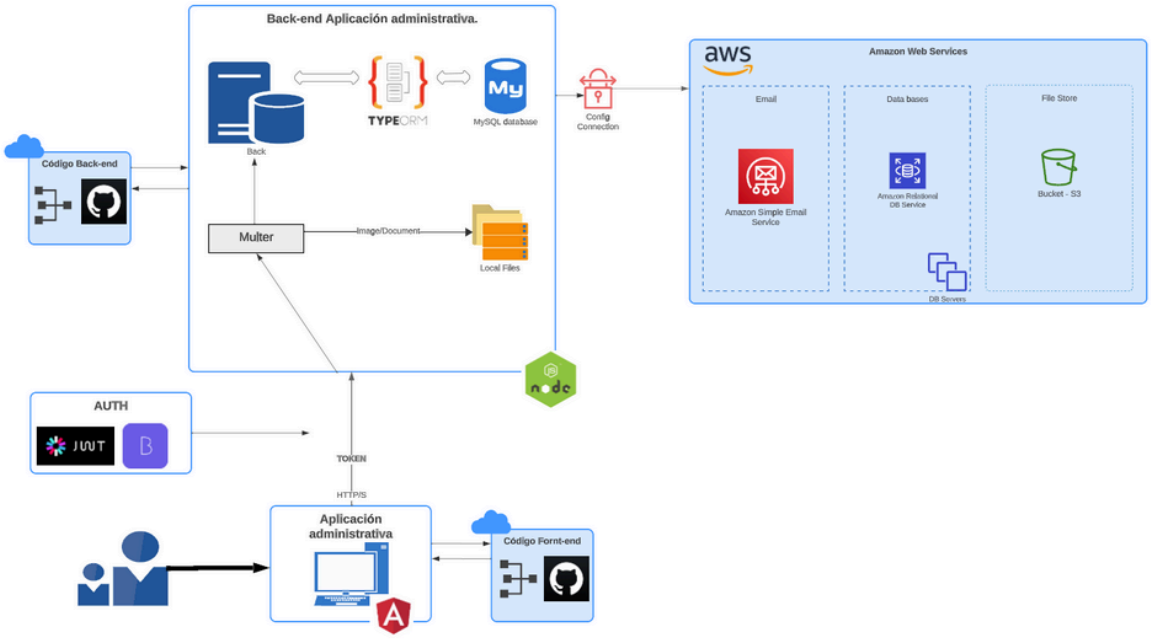


***Figura 22**. Diagrama sistema administrativo final.*

Tras la implementación, obtenemos el resultado ilustrado en las figuras: Figura 22 y Figura **23**.

4.1. Elección de las tecnologías

4.1.1. Tecnologías Front-end

Al momento de emprender la construcción de una aplicación web, se nos presenta una amplia y diversificada gama de librerías y frameworks a nuestro alcance, los cuales desempeñarán un papel fundamental en todo el transcurso del proceso. Tal y como se ha mencionado en la sección [2.1.2 Frameworks y bibliotecas], destacan tres opciones: Angular, React y Vue.js.

En el transcurso de las líneas siguientes, procedemos a efectuar una comparación de las características de cada uno de estos entornos de desarrollo:

Angular:

Figura 24. *Logo de Angular:*
https://es.wikipedia.org/wiki/Angular_%28framework%29

- Ventajas:
  - Angular es creado y mantenido por Google.
  - Angular ofrece una increíble estructura utilizando el patrón de diseño MVVM [13], el cual ayuda a conseguir un óptimo Routing, una inyección de dependencias y otras funcionalidades que veremos en el apartado de implementación.
  - Angular es modular y divide los componentes en chunks, es decir, agrupa la lógica en paquetes, lo que brinda una mayor optimización de los tiempos de carga y una mejora del rendimiento.
  - El uso del patrón MVVM aparte de acelerar el proceso del desarrollo, ayuda con la propagación automática del estado de la aplicación con el ***two-way binding.***
  - Angular es usado a nivel empresarial y con grandes aplicaciones gracias a su gran ecosistema y comunidad.
  - La utilización de TypeScript aporta una mejora en la calidad del código, sobre todo con la verificación de tipos, detección y corrección de errores.

- Desventajas:
  - Una de las principales desventajas de Angular es su difícil curva de aprendizaje, ya que exige más tiempo para dominarlo y requiere de habilidades básicas.
  - Si tu proyecto necesita una funcionalidad ***out-of-the-box***, es decir, que no viene directamente con el framework, puede producir un **bundle** pesado generando una aplicación más lenta.

React:
Debido a su esencia como biblioteca, React no presenta un diseño arquitectónico predefinido, dejando la estructura de la aplicación a la discreción del usuario. En otras palabras, React no incorpora de forma inherente un patrón de diseño como el Modelo-Vista-Controlador (MVC).

- Ventajas
  - El propósito de React es poder usarlo de manera rápida creando componentes reutilizables.
  - Una de las novedades de React es el uso de un virtual ***DOM [15]***, es decir, copia en la memoria el DOM actual del navegador, permitiendo realizar actualizaciones en la interfaz del usuario de manera eficaz.
  - Este tipo de DOM ayuda al ***tree-shaking [16]*** de React para optimizar el código de la aplicación reduciendo el tamaño del paquete final de JavaScript.

- Desventajas
  - Como hemos mencionado anteriormente, React no incluye una arquitectura para organizar las carpetas, si un usuario no dispone de conocimiento sobre las arquitecturas limpias puede encontrarse con dificultades a medida que el proyecto crece.
  - Si tenemos una aplicación compleja o mal optimizada, podremos notar problemas de rendimiento.
  - Toda la responsabilidad de instalar paquetes esenciales para el correcto funcionamiento de la aplicación se delega al usuario.

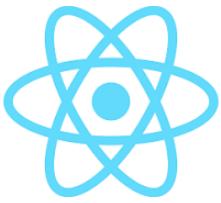


Figura 25. *Logo de React:*
https://commons.wikimedia.org/wiki/File:React-icon.svg

Una vez vistos los aspectos positivos y negativos de cada framework, nos hemos decantado por Angular para la creación de nuestro proyecto, ya que nos permite trabajar de forma modular, tiene un sistema de carga de componentes sobre la demanda, la estructura del proyecto nos viene ya marcada, y lo más importante, una vez acabada de crear la aplicación web la podremos convertir en una aplicación móvil con la tecnología PWA.

4.1.2. *Tecnologías Back-end*

En la parte del back-end nos encontramos con aquellas tecnologías que se ejecutan en el lado del servidor tanto para crear servidores web como bases de datos.
El framework más utilizado en el lado del back-end en los últimos años y que se encuentra en constante crecimiento es Node.js, todo esto es debido a las múltiples ventajas que ofrece, las cuales hemos comentado previamente en la sección de Frameworks:

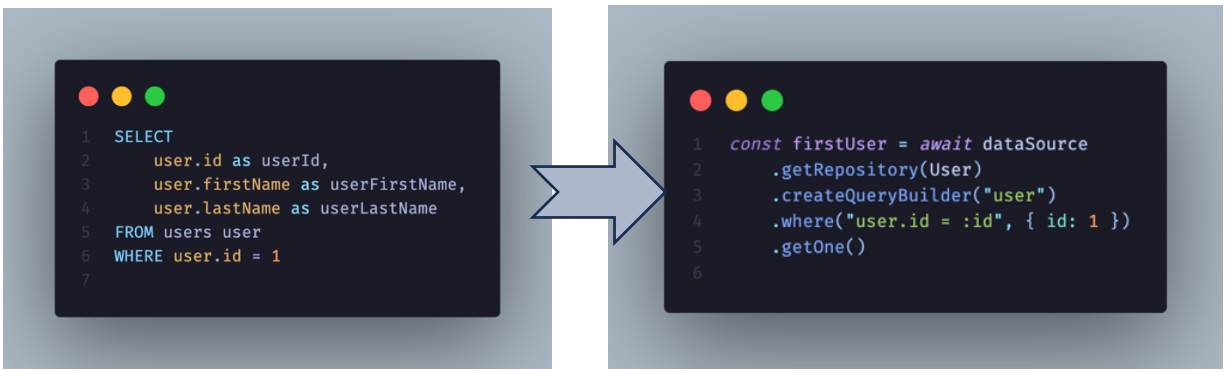
4.1.2.1. *TypeScript*

TypeScript [38] ha sido desarrollado por Microsoft el año 2012. Este nuevo lenguaje nació como una extensión al lenguaje de programación más utilizado en el desarrollo web que es JavaScript. El principal motivo de este lenguaje es mejorar todas aquellas limitaciones que este manifestaba.
Para aprovechar al máximo el tipado de los datos, la robusteza, flexibilidad y, sobre todo, los modelos de objetos que se crearán en Angular, usamos TypeScript también con Node.js

Figura 26. *Logo de TypeScript: https://blog.toothpickapp.com/top-10-things-to-know-about-typescript/*

4.1.2.2. *TypeORM*

TypeORM [17] es una librería de Object-Relational Mapping (ORM) para TypeScript que se utiliza en el desarrollo aplicaciones basadas en las bases de datos relacionales. Esta librería nos permite mapear objetos a una tabla en una base de datos, de esta forma nos ahorramos tiempo creando consultas para crear dichas tablas.
Mediante TypeORM podemos tratar las tablas como entidades (clases), y en consecuencia ahorramos tiempo escribiendo código SQL manualmente. También permite realizar migraciones de una manera fácil mediante scripts.
Sin duda, una de las características de TypeORM más destacable es la manera de realizar consultas legibles y mantenibles



4.1.2.3. Cloud computing:

La manera tradicional de trabajar con los servidores consiste en invertir una gran suma de dinero en el equipo físico (Racks), alojamiento de estos (alquiler del local), refrigeración y, sobre todo, en un equipo de seguridad para proteger a estos (tanto a nivel de ciberseguridad como ataques físicos), sin embargo, con el cloud computing tendremos acceso a una gran selección de servidores de manera rápida y sencilla.

Para cumplir con los requisitos de nuestro cliente y cubrir al máximo sus necesidades, debemos utilizar diferentes servicios que nos ofrece la nube.

Figura 27. *Figura de los tres competidores Cloud: https://trends.google.com/trends/explore?date=today%205-y&geo=ES&q=%2Fm%2F04y7lrx,%2Fm%2F0105pbj4,%2Fm%2F0rznzt1*

En la Figura 27, podemos examinar los tres principales competidores en el sector de cloud que son Amazon Web Service (AWS), Microsoft Azure y Google Cloud Platform.
A simple vista podemos ver que AWS entre el año 2018 y 2020 se ha mantenido constante, en cambio, en diciembre del año 2021 ha experimentado un aumento considerado de interés, el cual, según el periódico ABC [18] es debido a la crisis del Covid-19.

Tras la introducción al cloud computing y el análisis de los principales competidores, nos hemos decantado por el uso de los servicios de AWS en este proyecto, ya que nos impulsará a reducir costos y responsabilidad.

Figura 28. *Logo de AWS cloud: https://unaaldia.hispasec.com/2020/08/descubierto-criptominer-integrado-en-amazon-aws-community-ami.html*

Figura 29. *Figura de los servicios de AWS: https://www.projectpro.io/article/aws-projects-ideas-for-beginners/453*

4.1.2.4. *Docker*

Antes de comenzar a explicar que es Docker y que ventajas nos ofrece, es necesario saber cuáles han sido las complicaciones que nos han llevado hasta este punto. Para ello, estudiamos la evolución de los modelos de despliegue.

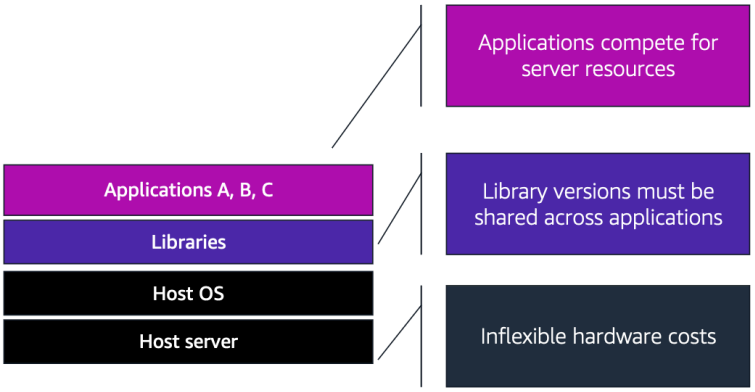


Figura 30. *Figura de servidores Bare-metal. Contenido del curso de AWS developer.*

La Figura 30 hace referencia a la estructura de un ***servidor bare-metal***, es decir, una máquina física tradicional sobre la cual se ejecutan las aplicaciones sin ningún tipo de virtualización.
El típico problema que encontramos con este tipo de servidor es: "¿Por qué en mi máquina funciona y en las demás no?", esto es debido a que durante el desarrollo de la aplicación se han ido instalando paquetes, configurando el entorno y al trasladarla a una nueva máquina, esta nueva carece de dicha configuración.

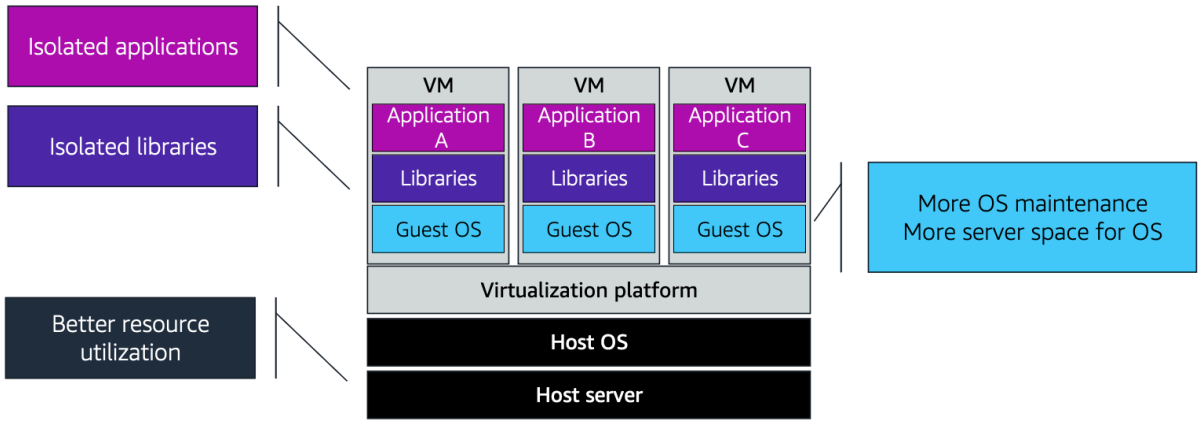


Figura 31. *Figura de servidores Bare-metal. Contenido del curso de AWS developer.*

La Figura 31 viene a resolver el problema comentado anteriormente, donde vemos ***un sistema de virtualización***, que aloja múltiples máquinas virtuales en un solo hardware. De esta forma, cada aplicación dispondrá de un entorno aislado con librerías y sistema operativo propio.
Cabe mencionar que el componente *Virtualization platform* es el encargado de la comunicación entre las diferentes máquinas virtuales (VMs) y el sistema operativo anfitrión.

Al cabo de un tiempo, los desarrolladores se han dado cuenta de que esta solución no es la definitiva, ya que cada VM tiene un sistema operativo propio, lo que lleva a una duplicación de recursos y como resultado, a unas máquinas pesadas.

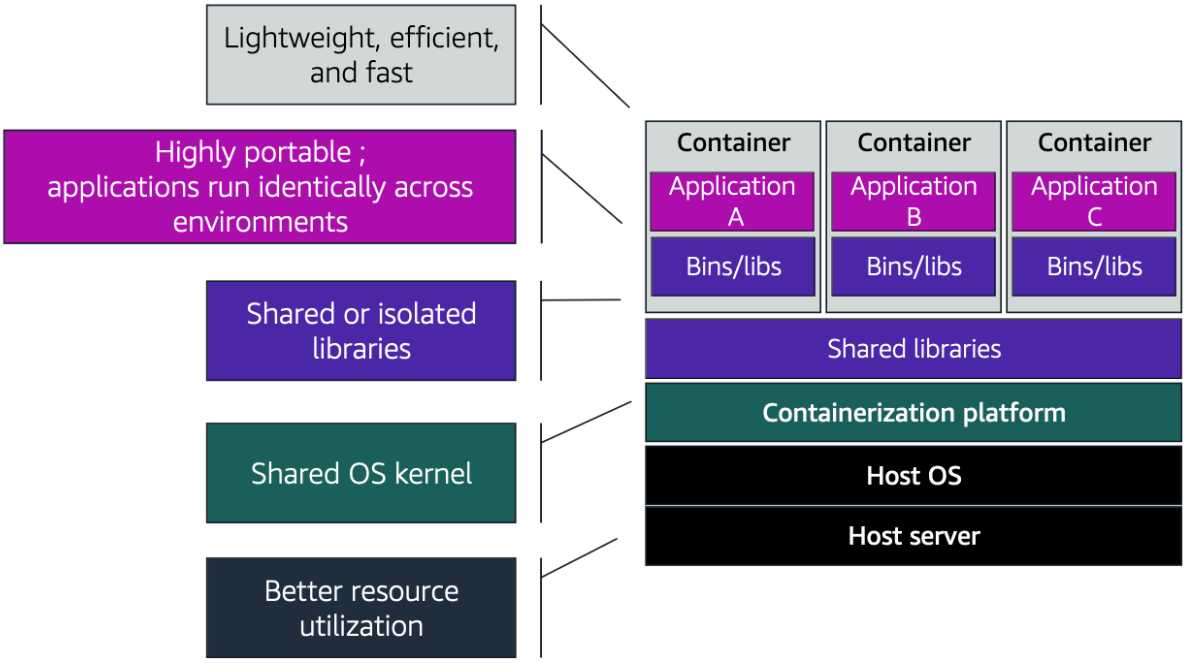


Figura 32. *Figura de servidores Bare-metal. Contenido del curso de AWS developer.*

Finalmente, llegamos a la solución de los *Containers*, donde un contenedor es una unidad ligera y portátil que contiene una aplicación, los ejecutables y las librerías necesarias para su correcto funcionamiento. De esta forma tenemos un sistema operativo instalado en la máquina, el cual es compartido entre los diferentes contenedores por el *Containerization platform*.
Incluso podemos tener librerías comunes a todos los contenedores, de este modo, evitaremos su instalación en cada uno de ellos.

Inmediatamente después de aclarar el concepto de contenerización, procederemos a explicar en qué consiste Docker.
Docker [19] es una plataforma de código abierto creado el año 2013, y diseñado para facilitar la creación, distribución y ejecución de las aplicaciones en contenedores.

Figura 33. *Logo de Docker: https://inlab.fib.upc.edu/ca/blog/docker-devops-tool.*

Las características de Docker son [20][21]:

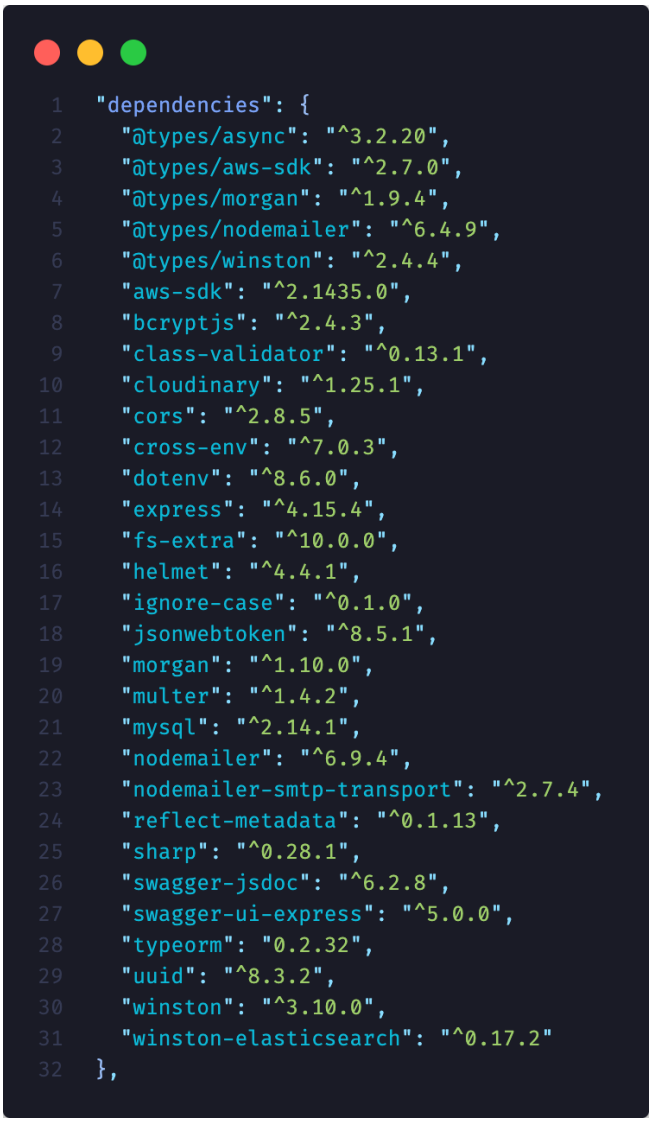
- **Portabilidad**: Los contenedores se pueden ejecutar en cualquier sistema operativo.
- **Ligereza**: Un contenedor pesa mucho menos que una VM, ya que comparte recursos con el sistema operativo.
- **Rapidez en el despliegue**: Las imágenes de un contenedor se crean y se distribuyen con facilidad.
- **Escalabilidad**: Docker es adecuado para un escalado horizontal, ya que podemos tener múltiples instancias idénticas de un contenedor y distribuir la carga entre ellos.
- **Orquestación**: Docker es fácil de integrar con otras herramientas de orquestación como Docker Swarm o Kubernetes.

4.2. *Implementación back-end aplicación administrativa*

En esta sección explicamos aquellas partes más destacadas en el desarrollo del back-end de la aplicación administrativa. Este back-end se encarga de brindar toda la información y ejecutar las funciones de la aplicación del front-end correspondiente.

4.2.1. *Paquetes instalados*

A continuación, detallamos los paquetes instalados más destacados:



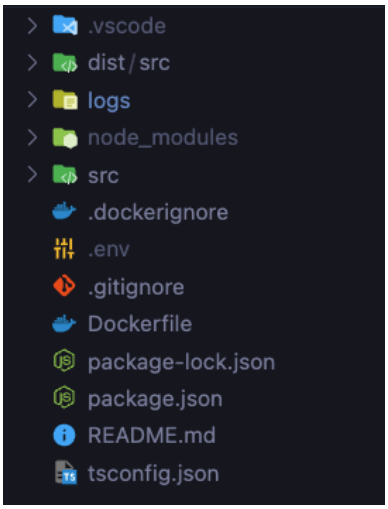
```
"dependencies": {
  "@types/async": "^3.2.20",
  "@types/aws-sdk": "^2.7.0",
  "@types/morgan": "^1.9.4",
  "@types/nodemailer": "^6.4.9",
  "@types/winston": "^2.4.4",
  "aws-sdk": "^2.1435.0",
  "bcryptjs": "^2.4.3",
  "class-validator": "^0.13.1",
  "cloudinary": "^1.25.1",
  "cors": "^2.8.5",
  "cross-env": "^7.0.3",
  "dotenv": "^8.6.0",
  "express": "^4.15.4",
  "fs-extra": "^10.0.0",
  "helmet": "^4.4.1",
  "ignore-case": "^0.1.0",
  "jsonwebtoken": "^8.5.1",
  "morgan": "^1.10.0",
  "multer": "^1.4.2",
  "mysql": "^2.14.1",
  "nodemailer": "^6.9.4",
  "nodemailer-smtp-transport": "^2.7.4",
  "reflect-metadata": "^0.1.13",
  "sharp": "^0.28.1",
  "swagger-jsdoc": "^6.2.8",
  "swagger-ui-express": "^5.0.0",
  "typeorm": "0.2.32",
  "uuid": "^8.3.2",
  "winston": "^3.10.0",
  "winston-elasticsearch": "^0.17.2"
},
```

-aws-sdk: Es el conjunto de herramientas que nos permite comunicarnos con la plataforma de AWS para gestionar diferentes recursos.
-bcryptjs: Usamos el paquete "bcryptjs" para crear el hash y la comparación segura de contraseñas de los usuarios.
-class-validatos: Este paquete nos ayudará a validar todos aquellos datos que un usuario nos pasará.
-cors: Este paquete nos sirve para controlar las solicitudes HTTP realizadas desde un origen hacia otro.
-dotenv: Se utiliza para cargar variables de entorno desde un fichero *.env*, de esta manera evitamos escribir directamente el valor de una variable en un fichero.
-express: Es un framework que nos proporciona herramientas para crear servidores web de manera rápida y sencilla.
-fs-extra: Es un módulo de Node.js que proporciona una API con más habilidades en comparación con "fs" para trabajar con el sistema de archivos.
-helmet [22]: Es un paquete que nos ayuda a proteger nuestra aplicación añadiendo encabezados y proporcionando capas adicionales de seguridad.
-jsonwebtoken [23]: Es una librería para poder trabajar con tokens JWT, es un estándar para la creación y verificación de tokens de seguridad.
-multer: Es una librería de middleware que se utiliza para manejar la carga de archivos.
-mysql: Para comunicarse con una base de datos tipo MySQL.
-nodemailer: Permite enviar emails.
-swagger: Nos permite documentar nuestras APIs.
-uuid: Es un paquete para generar cadenas de caracteres únicos para identificar objetos.
-winston: Nos proporciona una forma flexible para generar logs.

4.2.2. Estructura del proyecto

En esta sección podemos notar como está estructurado nuestro proyecto, seguidamente, analizamos cada una de estas:

Directorios:



•**dist**: La carpeta *dist* alojará el código TypeScript compilado a JavaScrpt para el entorno de producción, debido a que TypeScript se usa en entorno de desarrollo.

•**logs**: Como su nombre indica, esta carpeta contiene los diferentes ficheros de los logs que nuestra aplicación irá generando, estos logs serán de gran utilidad a la hora de monitorizar lo que pasa en esta misma.

•**node_modules**: Es el directorio que contiene todas las librerías que nuestra aplicación necesita para funcionar. Estas librerías se instalan ejecutando en comando *npm install* basándose en la declaración del fichero *package.json.*

•**src**: Esta carpeta contiene todo el código fuente de la aplicación implementada.

Ficheros:

- **.dockerignore**: En este fichero declaramos aquellas carpetas que no se desea que se incluye en la creación de un contenedor Docker.
- **.gitignore**: El mismo concepto se aplica con el fichero *.gitignore*, donde se declara aquel contenido que no se desea subir a un repositorio Git, por ejemplo: [.vscode/ node_modules/ build/ tmp/ temp/ .env].
- **.env**: Se trata del fichero que abarca las variables de entorno de nuestra aplicación.
- **Dockerfile**: Es el fichero de configuración que utiliza Docker para construir una imagen del contenedor basándose en las instrucciones que contiene este mismo.
- **package.json**: También se trata de un fichero de configuración utilizado por Node.js para definir aquellas dependencias, script y la información relacionada con la aplicación creada.
- **tsconfig.json**: Este último, también contiene la configuración utilizada en este proyecto para especificar las opciones del compilador TypeScript, es decir, permite especificar como se compila el código de TypeScript a JavaScript.

4.2.3. *Funcionalidad*

Una vez analizado la estructura general del proyecto, ahora nos centramos en la funcionalidad y en destacar las estrategias implementadas para conseguir un correcto funcionamiento.

```
import { App } from "./app";

async function main() {
  const app = new App(process.env.PORT || 5001);
  await app.listen();
}
main();

```

Como punto de entrada a la aplicación tenemos al fichero *index.ts,* el cual contiene una función *main* que crea una instancia de la clase App.

4.2.3.1. *App.ts*

La clase App que se instancia, se encuentra en el fichero *app.ts*, y contiene la siguiente configuración:

```
export class App {
  private app: Application;
  private swagger_config: any;
```

Como variables privadas de la clase, tenemos *app* y *swagger_config*, la variable *app* es de tipo *Application* que pertenece al paquete *express* mientras que *swagger_config* contiene la configuración necesaria para documentar nuestra Api.

```
constructor(private port?: number | string) {
    this.app = express();
    this.settings();
    this.middlewares();
    this.routes();
    this.dbConextion(ormconfig);
  }
```

Como es de costumbre, cada clase dispone de un constructor, en este caso recibe como parámetro el puerto.
El constructor asigna una instancia a nuestra variable *app*, y ejecuta una serie de funciones que detallaremos a continuación.

La primera función que veremos es *setting*, en la cual configuramos el puerto que expondrá la aplicación y definimos un token personalizado para el middleware del registro Morgan.

```
settings() {
    this.app.set("port", this.port || process.env.PORT || 3000);
    morgan.token(
      "message",
      (req: Request, res: Response) => res.locals.errorMessage || ""
    );
  }
```

```
middlewares() {
  this.app.use(cors());
  this.app.use(helmet());
  this.app.use(express.json());
  this.swagger_config = swaggerJsDoc(options);

  this.app.use(
    morgan(
      ":method :url :status :res[content-length] - :response-time ms :message",
      {
        stream: {
          write: (message) => logger.http(message),
        },
      }
    )
  );
}
```

La función *middlewares* en primer lugar configura los cors añadiendo así los encabezados necesarios para permitir o restringir las solicitudes entre diferentes máquinas (orígenes).
La segunda configuración tiene que ver con *hamlet* agregando diferentes encabezados para mejorar la seguridad de esta.

Para permitir a la aplicación manejar y procesar los datos en formato de JSON debemos configurar el middleware de nuestra aplicación añadiendo *express.json*.

```
export const options = {
  definition: {
    openapi: "3.0.0",
    info: {
      title: "API de la aplicación administrativa.",
      version: "1.0.0",
      description: "Descripción",
    },
    servers: [
      {
        url: "http://localhost:5000",
      },
    ],
  },
  apis: ["./src/routes/*.ts"],
};
```

Incluso para que nuestra aplicación tengo constancia de la librería *Swagger*, debemos adjuntarle una serie de configuración que tiene esta forma. En esta configuración, indicamos la versión de la aplicación, título, descripción, la *url* para acceder a la documentación de esta, y últimamente, indicamos las rutas que las que disponemos.

Para acabar con la función *middlewares*, configuramos el que viene a ser el *logger* de nuestra aplicación.

```
routes() {
  this.app.use("/", routes);
  this.app.use(
    "/docs",
    swaggerUI.serve,
    swaggerUI.setup(this.swagger_config)
  );
}
```

La siguiente función para comentar es *router* donde le hacemos saber a nuestra aplicación que rutas tenemos.
En este caso, tenemos una ruta raíz "/" que apunta a un fichero llamado *routes/index.ts* de esta forma, mantenemos limpio la clase *App*.
La otra ruta *"/docs"* es donde accedemos a la documentación de la API.

```
async dbConnection(ormconfig) {
  try {
    await createConnection(ormconfig);
  } catch (e) {
    console.log(e);
  }
}
```

```
const dotenv = require("dotenv");
dotenv.config();

const ormconfig = {
  type: "mysql",
  host: process.env.db_container,
  port: 3306,
  username: "root",
  password: process.env.db_pass,
  database: "tfg_db_dashboard",
  synchronize: true,
  logging: false,
  entities: ["src/entity/**/*.ts"],
  migrations: ["src/migration/**/*.ts"],
  subscribers: ["src/subscriber/**/*.ts"],
  cli: {
    entitiesDir: "src/entity",
    migrationsDir: "src/migration",
    subscribersDir: "src/subscriber",
  },
};

export default ormconfig;
```

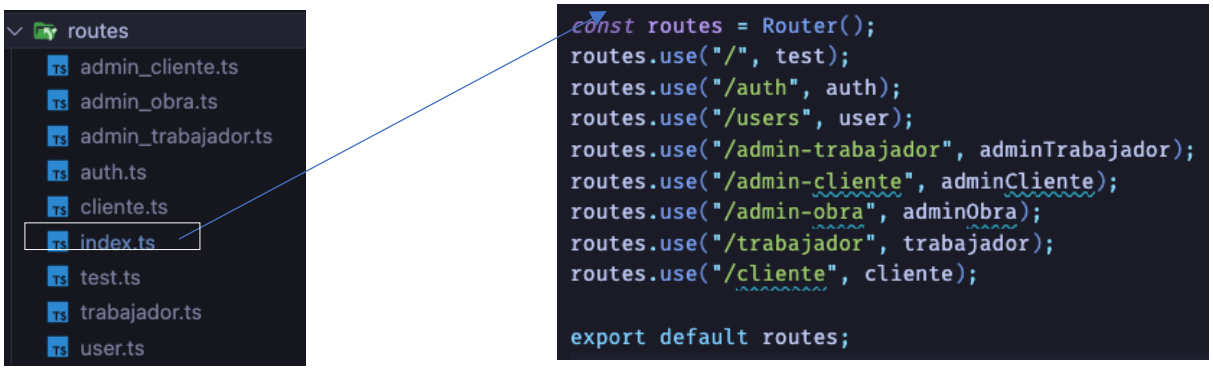
La función *dbConnection* se utiliza para poder establecer una conexión con la base de datos, la configuración de esta se encuentra en otro fichero (captura de la derecha).

```
async listen() {
    await this.app.listen(this.app.get("port"));
    console.log("server on port", this.app.get("port"));
  }
```

Para acabar de describir las funciones de la clase *App*, tenemos la función *listen*, la cual se encarga de poner a la escucha nuestra aplicación, de esta forma, empezará a resolver peticiones.

4.2.3.2. *Rutas*

En una *API* para acceder a cada recurso este tiene que estar asignado a una ruta (endpoint), a continuación, vemos aquellas rutas que forman nuestra aplicación.



En el directorio *router* se encuentran todas las rutas de la aplicación, es decir, al acceder a la ruta raíz "/", esta nos envía al fichero *index.ts* el cual tiene otras rutas que cuelgan de esta, y cada ruta al mismo tiempo apunta a otro fichero de rutas.
Por ejemplo, si accedemos a la ruta *"/admin-obra*" nos enviara al *adminObra* que tiene esta forma:

```
const router = Router();
router.get("/", [checkJwt, checkRole(["admin"])], AdminObra.getAllWorks);
router.get("/:id", [checkJwt, checkRole(["admin"])], AdminObra.getByWorkId);
router.patch("/:id", [checkJwt, checkRole(["admin"])], AdminObra.editWork);
router.post("/", [checkJwt, checkRole(["admin"])], AdminObra.newWork);
router.delete("/:id", [checkJwt, checkRole(["admin"])], AdminObra.deleteWork);
router.get(
  "/worker-to-work/:id",
  [checkJwt, checkRole(["admin"])],
  AdminObra.worker_to_work
);
router.post(
  "/worker-to-work/",
  [checkJwt, checkRole(["admin"])],
  AdminObra.add_worker_to_work
);
router.delete(
  "/worker-to-work/:idObra",
  [checkJwt, checkRole(["admin"])],
  AdminObra.delete_worker_to_work
);

router.post(
  "/image",
  multer.single("image"),
  [checkJwt, checkRole(["admin"])],
  AdminImage.uploadImage
);

router.post(
  "/image/publish/:id",
  [checkJwt, checkRole(["admin"])],
  AdminImage.publishImage
);
. . .
```

De esta manera vamos formando las diferentes rutas. Al llegar al final de cada ruta, esta ejecuta una función que realizará una lógica de negocio.

Nota: si nos fijamos en las rutas anteriores podemos observar que estas tienes un ***Guard***, es el encargado de comprobar si el usuario que ejecuta una ruta es quien debe ser.

4.2.3.3. *Auth*

En una aplicación que comparte recursos entre diferentes usuarios, una de las funcionalidades más esencial es el control del acceso a estos. En este apartado vemos cómo autenticamos a nuestros usuarios y sobre todo como restringimos el acceso a las rutas.

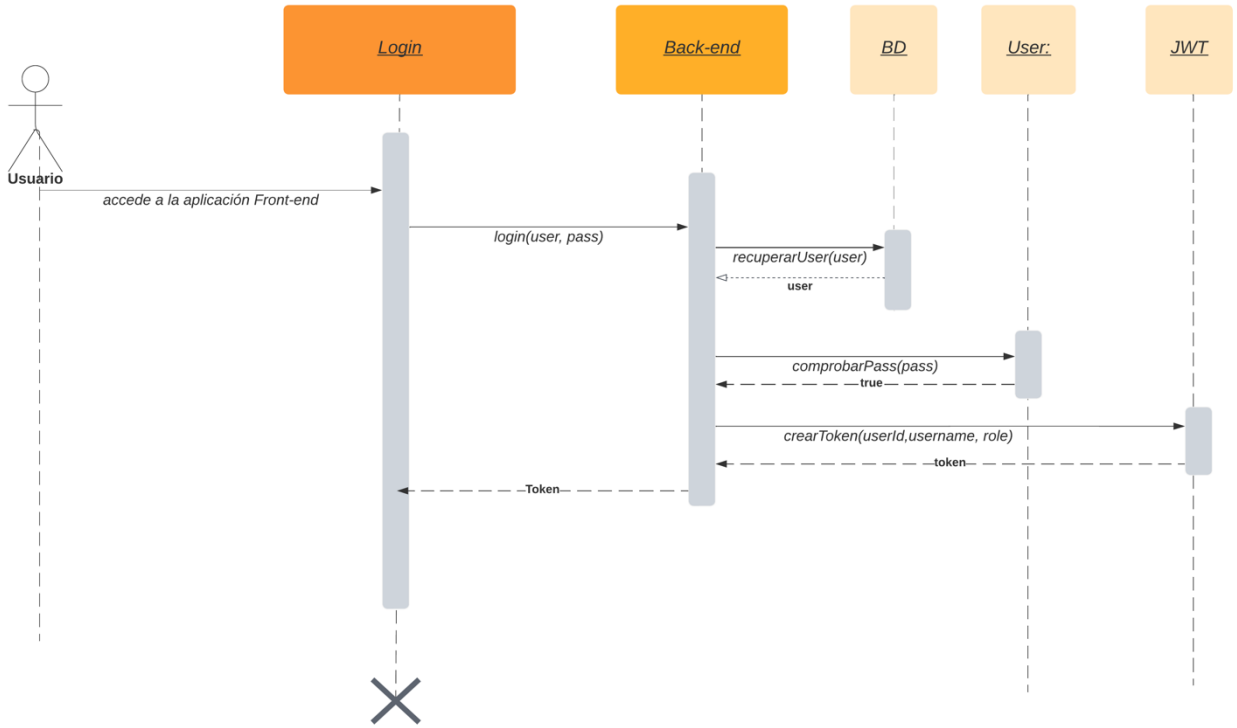
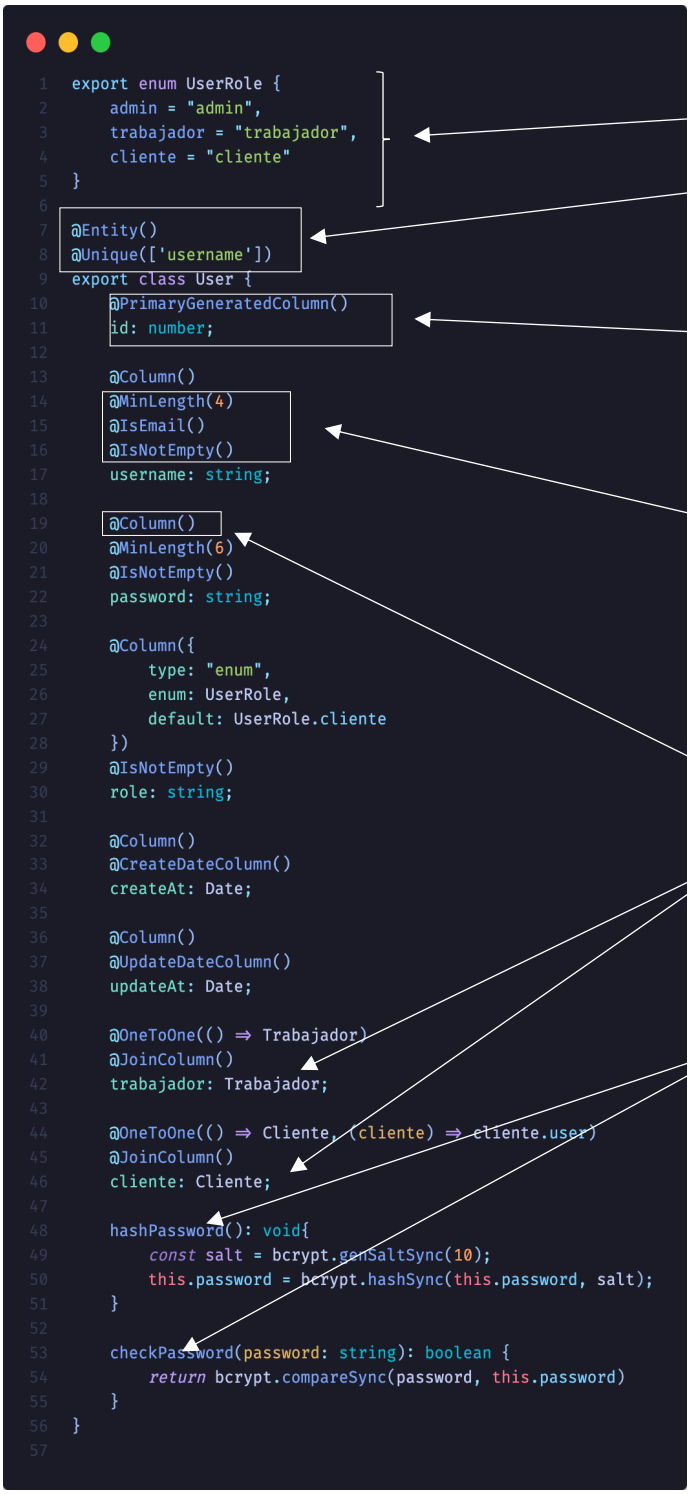


Figura 34. *Diagrama de la autenticación. Elaboración Propia*

Antes de comenzar con el código vamos a ver el flujo que transcurre entre el front-end y back-end para autenticarse y así para poder acceder a recursos.
En la Figura 34 podemos notar como el usuario accede a la página de *login* donde introduce sus credenciales, una vez introducido los datos, estos serán enviados a la función *login* del back-end. El servidor tras recibir los datos primero debe recuperar el usuario almacenado en la base de datos, si el usuario existe, se procederá a comprobar su contraseña, por lo cual debe aplicar un *hash* a la contraseña introducida en texto claro. Si ambos *hashes* coinciden, significa que las credenciales son correctas. Por último, crearemos un *token* con los datos necesarios y lo enviamos de vuelta al front-end.

4.2.3.4. *Entidad usuario*

Como hemos mencionado en el apartado de TypeORM, las tablas de la base de datos se crean a parir de una entidad, donde la entidad es una clase que tiene una serie de relaciones y atributos.



```
export enum UserRole {
    admin = "admin",
    trabajador = "trabajador",
    cliente = "cliente"
}

@Entity()
@Unique(['username'])
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    @MinLength(4)
    @IsEmail()
    @IsNotEmpty()
    username: string;

    @Column()
    @MinLength(6)
    @IsNotEmpty()
    password: string;

    @Column({
        type: "enum",
        enum: UserRole,
        default: UserRole.cliente
    })
    @IsNotEmpty()
    role: string;

    @Column()
    @CreateDateColumn()
    createAt: Date;

    @Column()
    @UpdateDateColumn()
    updateAt: Date;

    @OneToOne(() => Trabajador)
    @JoinColumn()
    trabajador: Trabajador;

    @OneToOne(() => Cliente, (cliente) => cliente.user)
    @JoinColumn()
    cliente: Cliente;

    hashPassword(): void{
        const salt = bcrypt.genSaltSync(10);
        this.password = bcrypt.hashSync(this.password, salt);
    }

    checkPassword(password: string): boolean {
        return bcrypt.compareSync(password, this.password)
    }
}
```

- Podemos asignar valores de tipo *enum* a una *columna*.
- Para indicar a TypeORM de que se trata de una tabla, usamos el decorador *@Entity* y además el nombre debe ser único
- Cada tabla debe tener un identificador, para ello usamos *@PrimaryGeneratedColumn* que nos crea un identificador autogenerado.
- Como hemos mencionado en varias ocasiones, una entidad es un objeto, por lo tanto, para añadir un objeto el usuario tiene que proporcionarnos dicha información, lo que implica realizar una validación.
- Para indicar las demás *columnas* de la tabla, usamos *@Column*.
- Un usuario puede ser de tipo Trabajador o Cliente, lo que implica tener que crear una relación, en este caso una relación de uno a uno.
- Una entidad también puede tener funciones propias, en esta situación tenemos dos: una para crear el *hash* de la contraseña utilizando un *salt* [2] y la otra, sirve para comprobar la contraseña actual del usuario a la que recibe por parámetro.

[2] Salt: es un valor único y aleatorio que se agrega a la contraseña antes de aplicar la función *hash* para evitar que dos contraseñas del mismo valor.

4.2.3.5. *Login*

```
static login = async (req: Request, res: Response) ⇒ {
    const { username, password } = req.body;

    if (!(username && password)) {
      return res.status(400).json({
        message: "Usuario y Contraseña obligatorios!",
      });
    }
    . . .
```

La función *login* recibe dos parámetros: el primero tipo *Request* y el segundo de tipo *Response*. Para enviar parámetro a este método, usamos el verbo *POST* del *HTTP*.
Mediante el cuerpo de la función (*req.body*) obtenemos el correo y la contraseña del usuario. La primera acción es comprobar si hemos recibido los datos, en caso contrario, enviamos un mensaje con un código de *status*.

```
. . .
const userRepository = getRepository(User);
let user: User;

try {
  user = await userRepository.findOneOrFail({ where: { username } });
} catch (e) {
  logger.error(e.message);
  return res
    .status(400)
    .json({ message: "Usuario o Contraseña incorrectos!" });
}
. . .
```

En el segundo paso, intentamos recuperar el usuario en la tabla *User,* para esto creamos un repositorio. Mediante el repositorio del usuario creamos una consulta de tipo *findOneOrFail* con la condición de que coincida el nombre del usuario con el objeto a recuperar.

```
. . .
if (!user.checkPassword(password))
  return res
    .status(400)
    .json({ message: "Usuario o Contraseña incorrectos!" });
. . .
```

Una vez recuperado el usuario, invocamos su función de *checkPassword* pasándole como argumento la contraseña recibida, lo que hará es hacer una comparación entre la contraseña que tenemos en la base de datos con la cadena de texto recibida, para ello utilizamos la librería ***bcrypt***.

```
bcrypt.compareSync(password, this.password)
```

```
  . . .
  const token = jwt.sign(
    { userId: user.id, username: user.username, role: user.role },
    config.jwtSecret,
    { expiresIn: "8h" }
  );

  res.json({
    message: "OK",
    token: token,
    role: user.role,
    user: user.id,
  });
};
```

Si las contraseñas coinciden significa que los datos introducidos son correctos, entonces debemos generar un *token*. Con ese propósito, usamos la función *jwt.sign* pasándole los datos a firmar y sobre todo, la clave secreta que tendremos solamente nosotros.
El *token* generado tendrá una fecha de validez de ocho horas, ya que la jornada laboral de un trabajador es de ocho horas.
Finalmente, enviamos al cliente (aplicación web) el *token* generado junto a la otra información.

Nota: la información que enviamos no es sensible.

4.2.3.6. *Multer*

Nuestra aplicación requiere un gran manejo de ficheros tanto documentos (PDF) como imágenes, lo que nos obliga a diseñar una estrategia óptima para poder brindar estos datos al cliente en menor tiempo, de lo contrario nuestra aplicación notará un retardo adicional.
En este punto entramos más en detalle sobre la funcionalidad que nos ofrece esta.
Multer es una librería de Node.js (middleware [24]) para manejar la subida de ficheros a un servidor a través de peticiones *HTTP*.

```
const storage = multer.diskStorage({
  destination: "uploads",
  filename: (req, file, callBack) => {
    callBack(null, uuidv4() + path.extname(file.originalname));
  },
});

export default multer({
  storage,
});

```

El fichero *multer.ts* es donde definimos el comportamiento que tendrá este paquete en nuestra aplicación. En primer lugar, con *multer.diskStorage* indicamos la carpeta (uploads) que alojará el archivo adjunto.
En un instante podemos tener dos objetos con el mismo nombre, por lo tanto, como consecuencia, nos llevaría a la perdida de estos, por esta razón debemos asignarles un nombre único.
Para llevar a cabo este renombramiento, usamos la función *callback* que se encarga de generar y asignar valores únicos al fichero (usando la librería *uuid*).

Llegados a este punto, tenemos que indicarle a Node.js cuales son aquellas funciones que recibirán ficheros, para ello inyectaremos la función anterior en las rutas correspondientes.

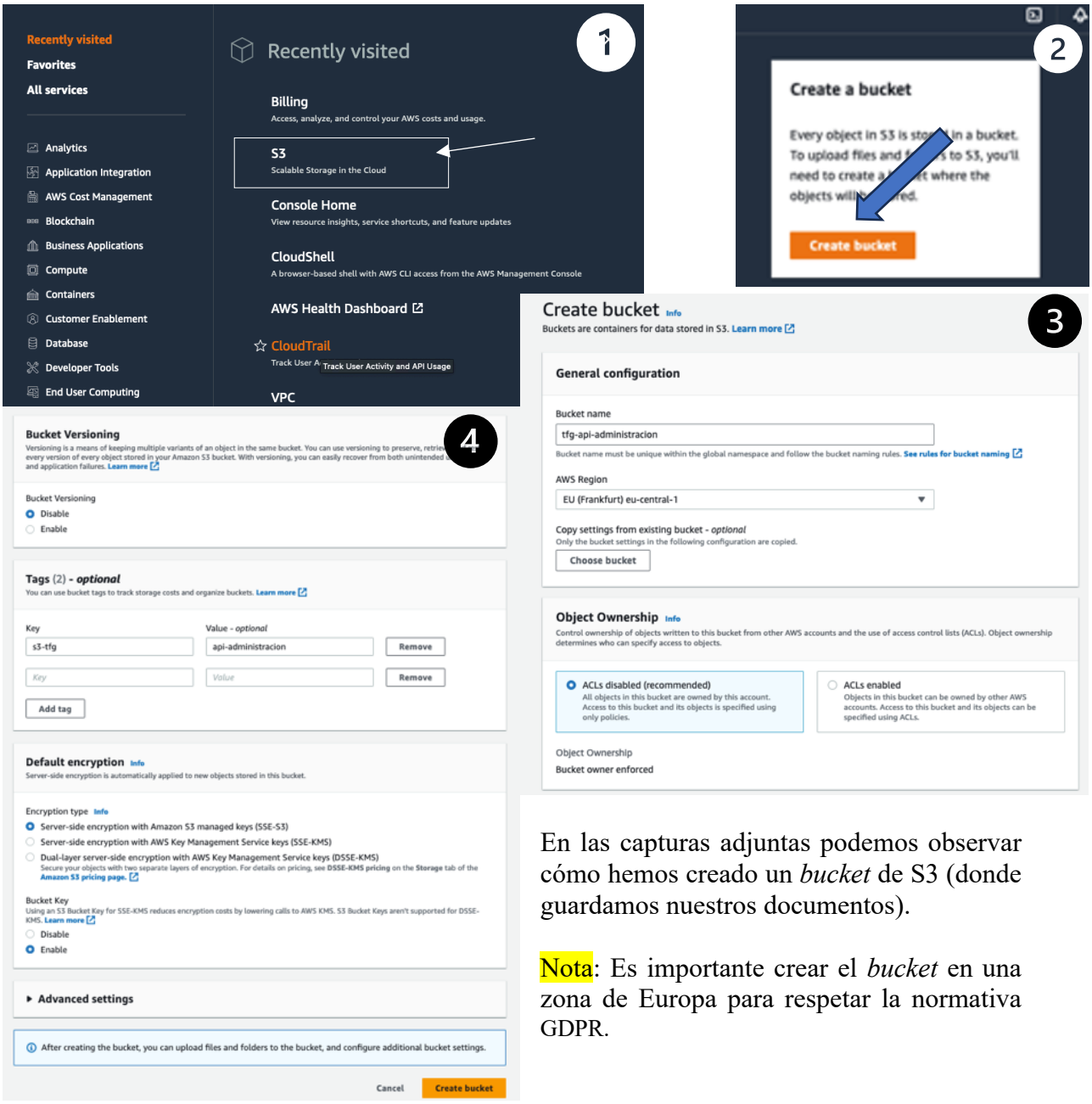
```
router.post(
  "/document",
  multer.single("document"),
  [checkJwt, checkRole(["admin"])],
  AdminDocumentos.uploadDocuments
);
```

```
router.post(
  "/image",
  multer.single("image"),
  [checkJwt, checkRole(["admin"])],
  AdminImage.uploadImage
);
```

4.2.3.7. *Amazon S3*

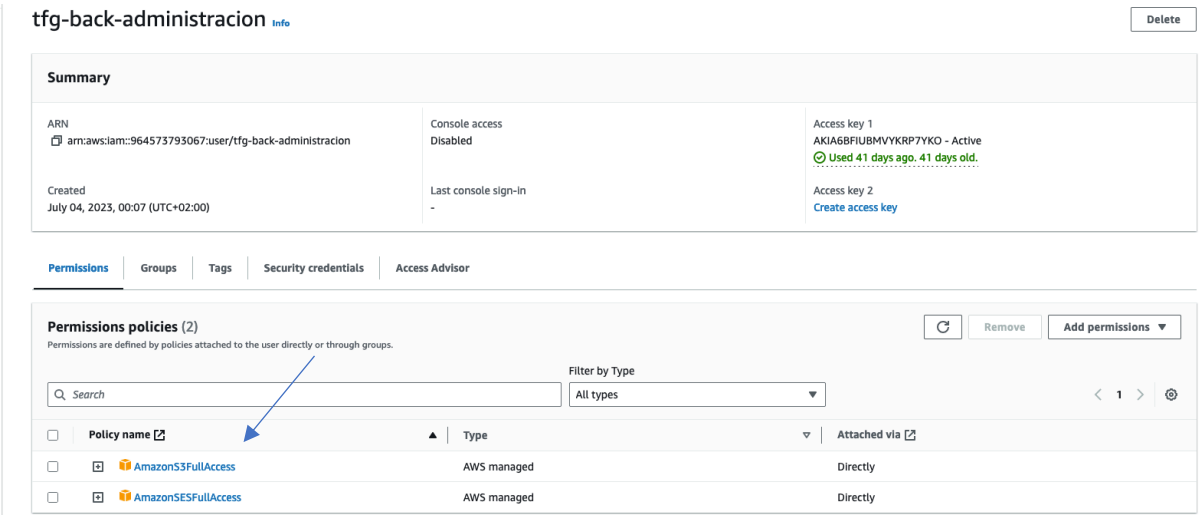
Como hemos comentado en el punto anterior, una gran parte del contenido de nuestra aplicación son los documentos e imágenes, debemos buscar una alternativa eficiente para almacenar estos ficheros.
La solución que implementamos en este proyecto consiste en subir estos ficheros a un servidor de almacenamiento en la nube altamente escalable y durable que será S3.
De esta manera evitamos guardar en la base de datos un objeto *blob* (*Binary Large Objec [25]*) que suelen tener hasta un tamaño de 65,535 bytes [26], en su lugar guardaremos una cadena de texto (*String*) que nos devolverá el servicio S3 al subir un fichero.

Antes de comenzar a utilizar S3, debemos crear una cuenta en la plataforma de *AWS*, crear usuario, y asignarle permisos para poder subir ficheros desde una aplicación mediante un *SDK*.



En las capturas adjuntas podemos observar cómo hemos creado un *bucket* de S3 (donde guardamos nuestros documentos).

Nota: Es importante crear el *bucket* en una zona de Europa para respetar la normativa GDPR.



Una vez creado un *bucket*, un usuario y asignado los permisos necesarios, ya podemos empezar a subir nuestros ficheros. Para ilustrar un ejemplo, nos basamos en la función *uploadDocuments* del rol administrador.

```
const AWS = require('aws-sdk');

  AWS.config.update({
    accessKeyId: process.env.AWS_ACCESS_KEY_ID,
    secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
    region: 'eu-central-1'
  });
```

Primero debemos configurar las credenciales de acceso de *AWS* para que la aplicación pueda acceder a S3.

Una vez situados dentro de la función, tenemos que crear una instancia al servicio S3 y establecer la configuración del *bucket* donde queremos subir el documento.

```
static uploadDocuments = async (req: Request, res: Response) => {
    s3 = new AWS.S3();

    const params = {
      Bucket: "tfg-api-administracion",
      Key: req.file.originalname,
      Body: req.file.path,
    };
```

```
. . .
    const { obrasId, directorio, tipo } = req.body;

    if (obrasId == null || obrasId == "" || req.file == null) {
      await fs.unlinkSync(req.file.path);
      res.json({ message: "Faltan argumentos!" });
    }

    if (req.file.mimetype != "application/pdf") {
      await fs.unlinkSync(req.file.path);
      return res
        .status(404)
        .json({ message: "No se admite este tipo de documentos!" });
    }
    . . .
```

Antes de invocar el método de S3, tenemos que realizar una serie de comprobaciones, tanto para mirar si recibimos toda la información como si el tipo del documento es el esperado.

Si hemos pasado las comprobaciones, entonces significa que ya podemos subir el fichero al *bucket* de S3.

```
. . .
    s3.putObject(params, (err, data) => {
      if (err) {
        logger.error(err);
        return res.status(code).json({ message: "No se ha podido subir el fichero!" });
      }
    });
    . . .
```

```
    . . .
    const docu_url = `https://tfg-api-administracion.s3.amazonaws.com/${req.file.originalname}`;

    const docu = new Documento();
    docu.obra = obrasId;
    docu.url = docu_url;
    docu.directorio = directorio;
    docu.tipo = tipo;

    const documentRepository = getRepository(Documento);
    documentRepository.save(docu);

    await fs.unlinkSync(req.file.path);

    return res.json({ message: "Documento guardado correctamente!" });
  } catch (e) {
    logger.error(e.message);
    await fs.unlinkSync(req.file.path);
    return res.status(code).json({ message: "Error al subir el fichero!" });
  }
};
```

Una vez subido el documento, continuamos con la ejecución creando y guardando el objeto en la tabla correspondiente.

4.2.3.8. *Amazon SES*

Una funcionalidad esencial para destacar es el sistema de correos electrónicos, ya que gracias a ello podemos notificar un evento de manera rápida. En este apartado comentamos como hemos implementado esta función con el servicio *SES* de Amazon.
Por ejemplo, el caso de uso que requiere enviar correo a un usuario es la función *createIncidence* del rol trabajador, puesto que cuando un trabajador crea una incidencia, aparte de guardarla en nuestra base de datos, debemos notificar al administrador vía correo de esta para agilizar el proceso. Seguidamente, tratamos el método *createIncidence* y cómo está implementado.

```
static crear = async (req: Request, res: Response) => {
  const { reason, urgency, obraId } = req.body;
  const incidencia = new Incidencia();
  incidencia.reason = reason;
  incidencia.date = new Date();
  incidencia.urgency = urgency;
  incidencia.obra = obraId;
  incidencia.viewed = false;

  const validationOpt = { validationError: { target: false, value: false } };

  const errores = await validate(incidencia, validationOpt);
  if (errores.length > 0) return res.status(code).json(errores);
  . . .

```

Después de recibir los parámetros enviados por el cliente (aplicación front-end), creamos una nueva instancia *Incidencia*. Cabe destacar que antes de almacenar este nuevo objeto, debemos validar los datos recibidos mediante el paquete *class-validator*.

Una vez realizado la validación, procedemos a recuperar la obra relacionada con la incidencia, y en caso de recibir un error, notificamos el mismo e interrumpimos la ejecución.

```
. . .
  let work;
  try {
    work = await getConnection()
      .getRepository(Obra)
      .createQueryBuilder("obra")
      .where("obra.id = :id", { id: obraId })
      .getOne();
  } catch (e) {
    logger.error(e.message);
    return res.status(code).json({ message: "Algo ha ido mal!" });
  }

  if (!work) return res.status(code).json({ message: "La obra no existe!" });
  . . .
```

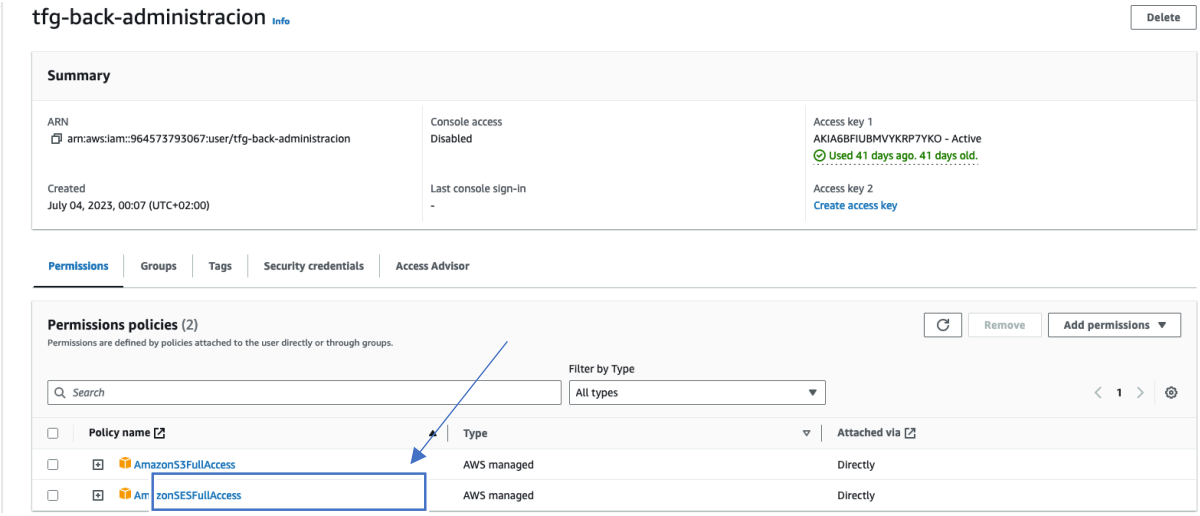
```
    . . .
    const userRepository = getRepository(Incidencia);
    try {
      await userRepository.save(incidencia);
      await sendEmail(obraId, work.name, reason);
    } catch (e) {
      return res
        .status(code)
        .json({ message: "Error a la hora de crear la incidencia." });
    }
    return res.send({ message: "Incidencia creada correctamente!" });
  };
```

Posteriormente, guardamos la incidencia en nuestra base de datos y enviamos el correo.

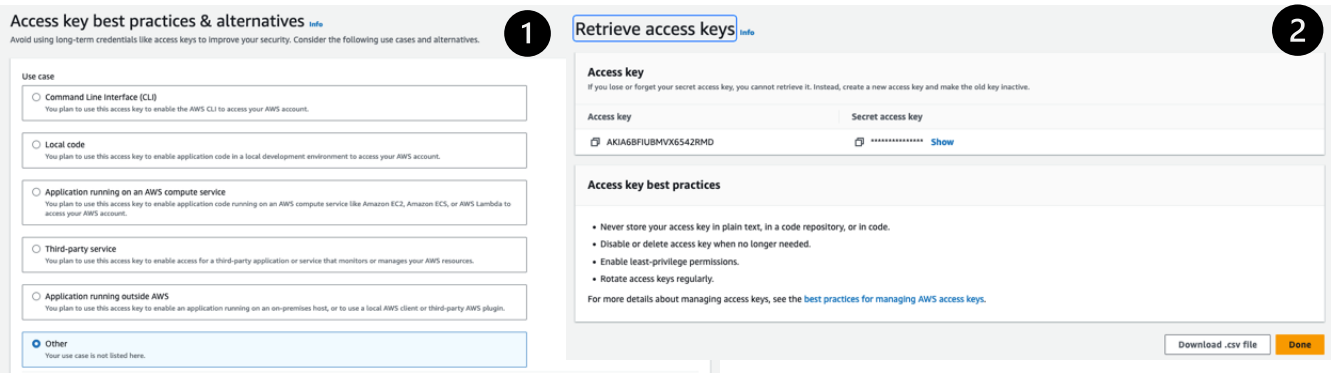
En las siguientes líneas, analizamos la función *sendEmail*.

Para poder utilizar el servicio *SES* de *AWS* debemos seguir unos pasos:

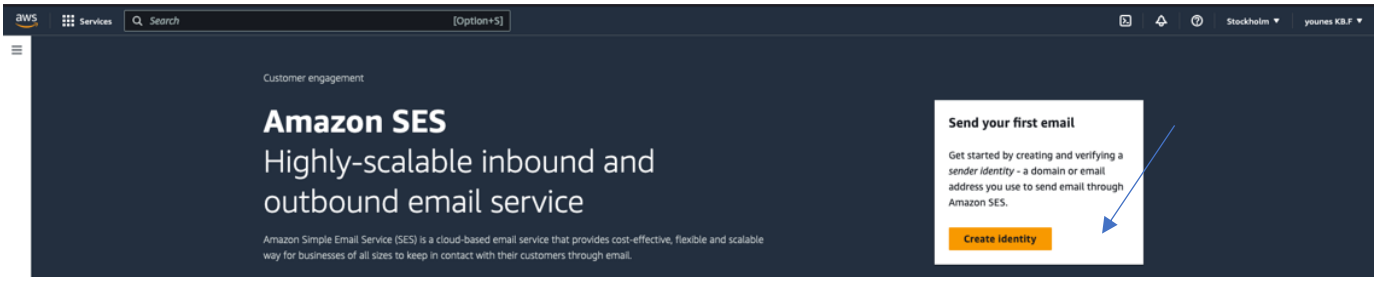
1. Primero debemos añadir permisos *SES* al usuario que tenemos en la plataforma.

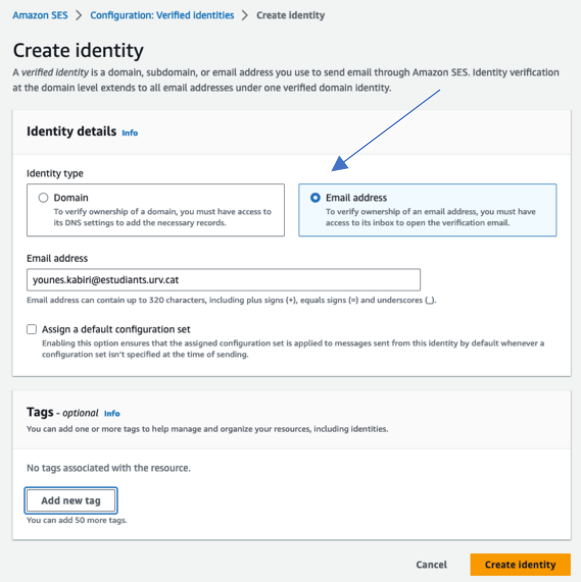


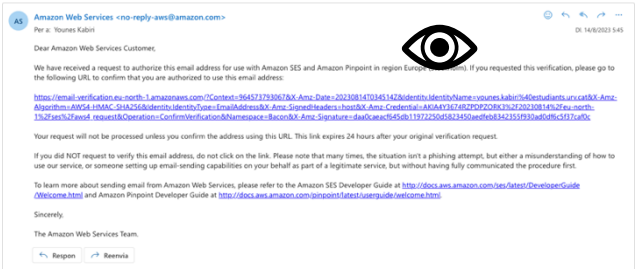
2. Una vez tenemos los permisos necesarios, creamos las credenciales para poder acceder al servicio *SES* desde nuestro back-end.



3. Damos de alta en la plataforma de *AWS* al correo que se encargará notificar al administrador cuando ocurre una incidencia.
   Después de añadir el correo es muy importante verificar esa cuenta, ya que de lo contrario no se enviará nada.







Tras haber realizado la configuración necesaria, ya podemos lidiar con el envío de correo desde nuestra aplicación back-end.

```
import * as nodemailer from "nodemailer";
import * as AWS from "aws-sdk";

const dotenv = require("dotenv");
dotenv.config();

AWS.config.update({
  accessKeyId: process.env.aws_email_access_key,
  secretAccessKey: process.env.aws_email_access_secret_key,
  region: process.env.aws_region, // Por ejemplo, 'us-west-2'
});

const transporter = nodemailer.createTransport({
  SES: new AWS.SES(),
});
```

Para enviar el email, como primer paso, debemos autenticarnos con el servidor de *AWS* utilizando las credenciales obtenidas en el paso anterior.
Usamos el *SDK* de Amazon junto con la librería *nodemailer* para llevar a cabo esta operación.

La función *sendEmail* recibe como parámetros tres argumentos: el identificador de la obra involucrada, su nombre y el mensaje a enviar.

```
export async function sendEmail(
  idObra: string,
  nombreObra: string,
  mensaje: string
) {
```

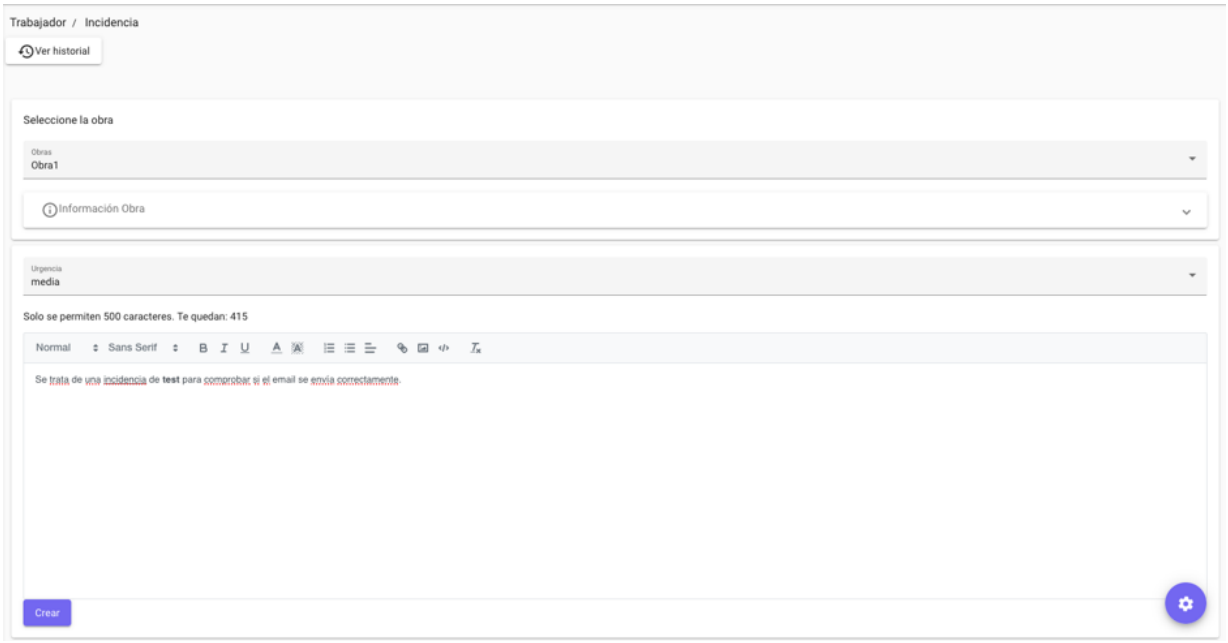
```
const html_body =
  `
      <body style="
          .contenedor{
              width:100%;height:100px;
          }
          .contenido{
              height:80px;
              width:10%;
              float:left;
              margin: 10px;
          }">

          <p>Este correo se ha enviado desde la aplicación administrativa, en la sección de incidencias.</p>
          <div class="contenedor">

              <div class="contenido">
                  <h4>Id obra:  ` +
  idObra +
  `</h4>
              </div>

              <div class="contenido">
                  <h4>Nombre obra:  ` +
  nombreObra +
  `</h4>
              </div>

              <div class="contenido">
                  <h4>Mensaje:</h4>
                  <p>` +
  mensaje +
  `</p>
              </div>
          </div>
      </body>
    `;
```

Para personalizar nuestro correo y añadirle algo de diseño, podemos adjuntar *HTML* con *CSS* en lugar de un simple texto.

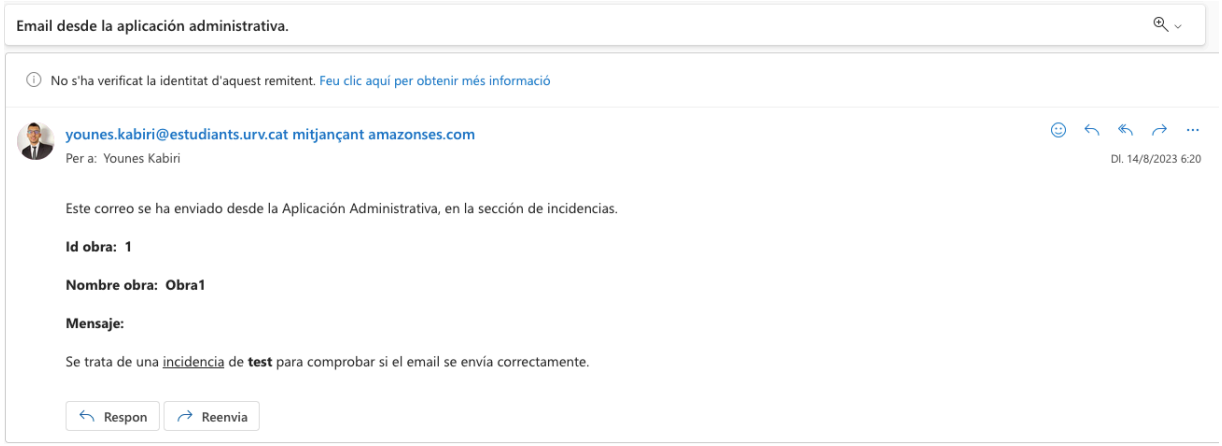
```
    . . .
    const mailOptions = {
      from: process.env.email,
      to: process.env.email_admin,
      subject: "Email desde la aplicación administrativa.",
      html: html_body,
    };

    try {
      const info = await transporter.sendMail(mailOptions);
      console.log("Correo enviado:", info.response);
      return true;
    } catch (error) {
      console.error("Error al enviar el correo:", error);
      return false;
    }
  }

```

Para acabar, en *mailOptions* definimos el que viene a ser el correo, remitente, destinatario, asunto y el *html* que contiene el mensaje a enviar.

Ejemplo de una incidencia:



Resultado:



4.2.3.9. *Guard*

Como hemos mencionado en la sección *Auth* es importante proteger los recursos, para esa labor implementamos lo que se conoce como *Guard*.
Un *Guard* es un *middleware* que tiene como función proteger una ruta de acceso no autorizado, es decir, solo los usuarios autorizados y con el rol requerido pueden acceder a un recurso. Con solo estar autenticado no te permite acceder a todas las funciones, puesto que un usuario con rol de cliente tendría acceso a funciones del administrador, y este comportamiento no es el deseado.

Para llevar a cabo la capa de seguridad comentada, desarrollamos dos funciones: *checkJwt* y *checkRole(role)*.

```
export const checkJwt = (req: Request, res: Response, next: NextFunction) => {
  const token = <string>req.headers["auth"];
  let jwtPayload;

  try {
    jwtPayload = <any>jwt.verify(token, config.jwtSecret);
    res.locals.jwtPayload = jwtPayload;
  } catch (e) {
    return res.status(401).json({ message: "No autorizado!" });
  }

  const { userId, username } = jwtPayload;
  const newToken = jwt.sign({ userId, username }, config.jwtSecret, {
    expiresIn: "8h",
  });
  res.setHeader("token", newToken);

  next();
};

```

La función *checkJwt* se encarga de comprobar la validación del *token* adjuntado en el encabezado de la solicitud. Mediante el método ***jwt.verify*** verificamos la autenticidad del *token*, es decir, verificar si este es válido (autenticidad) y no modificado (integridad).
Si la verificación tiene éxito, el objeto ***jwtPayload*** contendrá los datos decodificados del *token* como el ID del usuario y el rol, los cuales son empleados para crear uno nuevo.

El hecho de renovar el *token* nos ofrece una serie de ventajas:

- La renovación automática, permite al usuario seguir utilizando la aplicación sin tener que iniciar sesión nuevamente cada vez que expire.
- La emisión de un nuevo *token* en lugar de extender la vida útil del original mantiene la seguridad de nuestra aplicación intacta. Si el original ha sido comprometido, su tiempo de validez se limita al tiempo recorrido entre las dos peticiones.
- Minimizar la ventana de ataques, es decir, si un *token* es robado o interceptado durante la transmisión, el atacante tendrá un tiempo limitado de uso.

La segunda función a explicar es el *checkRole*:

```
export const checkRole = (roles:Array<string>) => {
    return async (req: Request, res: Response, next: NextFunction) => {

        const { userId } = res.locals.jwtPayload;
        const userRepository = getRepository(User);

        let user: User;

        try {
            user = await userRepository.findOneOrFail(userId);
        }catch(e) {
            return res.status(code).json({ message: 'No autorizado!' });
        }


        const { role } = user;
        if(roles.includes(role))
            next();
        else
            res.status(code).json({ message: 'No autorizado!' });
    }
}
```

Esta función se ejecuta justo después de la primera (*checkJwt*), de esta forma nos aseguramos de la validez e integridad de los datos que empleamos. En el primer paso se procede a extraer el identificador del usuario, el cual nos sirve para recuperar el registro del usuario en la base de datos. Tan pronto obtenido el usuario, comprobamos entre su rol y el rol del *token*, si coinciden, le permitiremos acceso, de lo contrario se le restringirá el mismo.

Finalmente, nos faltará inyectar estas funciones en nuestras rutas, es importante mencionar que el método **checkJwt** es igual para todas ellas, mientras que **checkRole** será diferente para las rutas de cada rol.

Role administrador

```
. . .
router.get(
  "/",
  [checkJwt, checkRole(["admin"])],
  AdminTrabajador.getAllWorkers
);

router.get(
  "/:id",
  [checkJwt, checkRole(["admin"])],
  AdminTrabajador.getWorkerById
);
. . .
```

Role trabajador

```
. . .
router.post(
  "/incidencia",
  [checkJwt, checkRole(["trabajador"])],
  TrabajadorIncidencia.crear
);

router.post(
  "/incidencia/mark-viewed",
  [checkJwt, checkRole(["trabajador"])],
  TrabajadorIncidencia.Seen
);
. . .
```

Role cliente

```
. . .
router.get(
  "/Obra/:id",
  [checkJwt, checkRole(["cliente"])],
  ClienteController.getObra
);

router.get(
  "/:id",
  [checkJwt, checkRole(["cliente"])],
  ClienteController.getClientById
);
. . .
```

4.2.3.10. Relaciones entre entidades

Además de las consultas de TypeORM que hemos ido viendo durante el desarrollo de este proyecto, en este apartado, mostramos más ejemplos de cómo elaboramos dichas consultas para obtener los datos necesarios en cada caso.

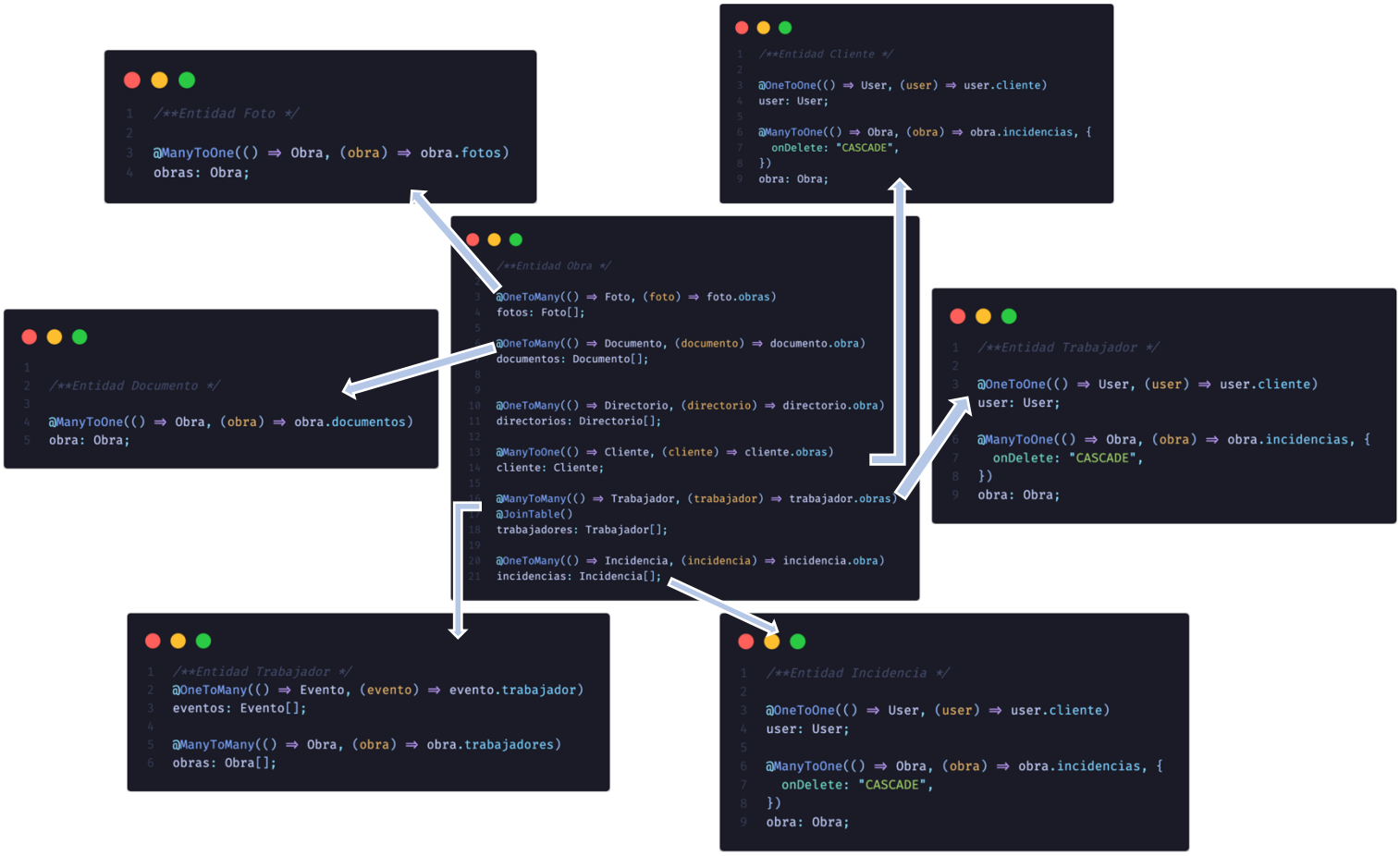
En lugar de utilizar SQL:

```sql
SELECT obra.*, cliente.*, trabajador.*
FROM obra
LEFT JOIN cliente ON obra.clienteId = cliente.id
LEFT JOIN trabajador ON obra.id = trabajador.obraId
WHERE obra.id = :id;
```

Usamos TypeORM y queda de la siguiente forma:

```
let work;
try {
  work = await getConnection()
    .getRepository(Obra)
    .createQueryBuilder("obra")
    .where("obra.id = :id", { id: id })
    .leftJoinAndSelect("obra.cliente", "cliente")
    .leftJoinAndSelect("obra.trabajadores", "trabajador")
    .getMany();
} catch (e) {
  logger.error(e.message);
  res.status(code).json({ message: "Obra no encontrada!" });
}
```

En vez de obtener los datos relacionados a una sola entidad, podemos observar cómo recuperamos la información del cliente y de los trabajadores utilizando *leftJoinSelect, where, y andWhere* pero para que sea posible tenemos que relacionar las entidades de esta forma:



En la clase *Obra* tenemos una serie de relaciones hacia otras entidades, al mismo tiempo, cada una de estas clases debe retornar el tipo de relación establecida dependiendo de la naturaleza de relación que nos interesa mantener.
En la imagen presentada, podemos observar de manera gráfica como se forman estas relaciones.

4.2.3.11. *Documentación*

Una funcionalidad muy importante además de la codificación y el *Testing* es la documentación, ya que nuestra *API* puede ser utilizada posteriormente por otros desarrolladores y la manera más eficaz de saber las rutas de las que dispone esta, su utilidad y, sobre todo, los parámetros que necesita es mediante la documentación. En este segmento, estudiamos el desarrollo que se ha llevado a cabo para documentar nuestros endpoints.

En el apartado *App.ts* hemos explicado la configuración que necesita el paquete *Swagger* para ser integrado en nuestra aplicación, a continuación, ilustramos el resto de los ajustes realizados. Como ejemplo, seleccionamos la ruta */auth* la cual tiene dos endpoints: */login* y */change-passwo*

```
/**
 * @swagger
 * tags:
 *  name: auth
 *  description: Session-related methods
 */
```

Primeramente, *Swagger* utiliza el formato *YAML* [27] para describir la estructura de nuestra documentación.
En esta captura de código podemos observar cómo creamos un *tag*, el cual será útil para agrupar diferentes rutas bajo un dominio.

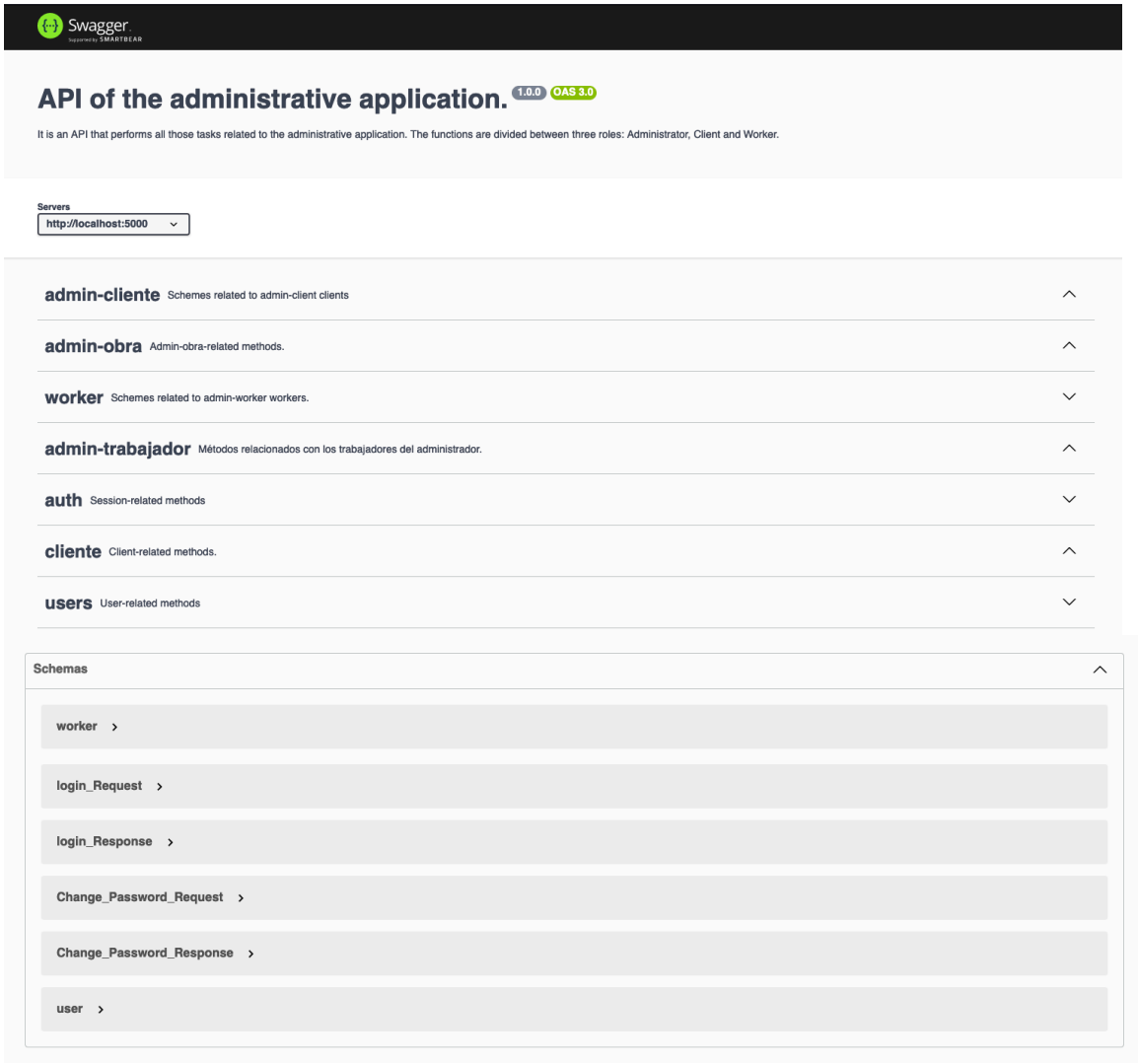
```
/**
 * @swagger
 * /auth/login:
 *   post:
 *     summary: Logging
 *     tags: [auth]
 *     requestBody:
 *         required: true
 *         content:
 *             application/json:
 *                 schema:
 *                     $ref: '#/components/schemas/login_Request'
 *     responses:
 *         200:
 *             description: Method for logging into the system
 *             content:
 *                 application/json:
 *                     schema:
 *                         $ref: '#/components/schemas/login_Response'
 */
```

En la figura anterior, podemos observar cómo indicar a *Swagger* los detalles de una ruta, sus parámetros, cuerpo y la respuesta que devolverán. Aparte de la ruta y la descripción, podemos notar la aparición de unos esquemas de tipo *JSON* que veremos seguidamente.

```
/**
 * @swagger
 * components:
 *  schemas:
 *    login_Request:
 *      type: object
 *      properties:
 *          email:
 *              type: string
 *          password:
 *              type: string
 *      example:
 *          email: user@gmail.com
 *          password: 1234
 *
 *
 *    login_Response:
 *      type: object
 *      properties:
 *        token:
 *          type: string
 *          description: generated token.
 *        role:
 *          type: string
 *          description: role of the user.
 *        user:
 *          type: number
 *          description: number id of the user
 *      example:
 *        token: eyJhbGciOiJIUzI1NiIsInR5cCI6 ...
 *        role: trabajador
 *        user: 1001
```

En la captura presentada, podemos ver la notación para declarar dos esquemas llamados *login_Request* y *login_Response*. En cada esquema, aparte de indicar los atributos que lo forman, también facilitamos un ejemplo de este.

Últimamente, para ver la documentación generada podemos acceder a la ruta especificada en la configuración previa, que es *http://localhost/5001/docs* (en entorno de desarrollo).



En esta captura podemos ver la información de la *API* como la versión, nombre o descripción. También se puede apreciar las rutas que tiene la aplicación, los esquemas y, sobre todo, si clicamos en cada zona, podemos obtener más información sobre cada una de estas.
En las siguientes capturas, podemos contemplar el resultado obtenido.

admin-cliente Schemes related to admin-client clients

admin-obra Admin-obra-related methods.

worker Schemes related to admin-worker workers.

- GET `/admin-trabajador/` get all workers.
- GET `/admin-trabajador/{id}` get Worker by Id.
- POST `/admin-trabajador/{id}` create new worker.
- PATCH `/admin-trabajador/{id}` modify worker.
- DELETE `/admin-trabajador/{id}` delete Worker by Id.

admin-trabajador Métodos relacionados con los trabajadores del administrador.

auth Session-related methods

- POST `/auth/login` Logging
- POST `/auth/change-password` Change Password

cliente Client-related methods.

users User-related methods

- GET `/users` getAllUsers
- GET `/users/:id` get User by Id.
- PATCH `/users/:id` modify User by Id.
- DELETE `/users/:id` delete User by Id.

auth Session-related methods

POST `/auth/login` Logging

Parameters | Try it out

No parameters

Request body required | application/json

Example Value | Schema

```
{
  "email": "user@gmail.com",
  "password": 1234
}
```

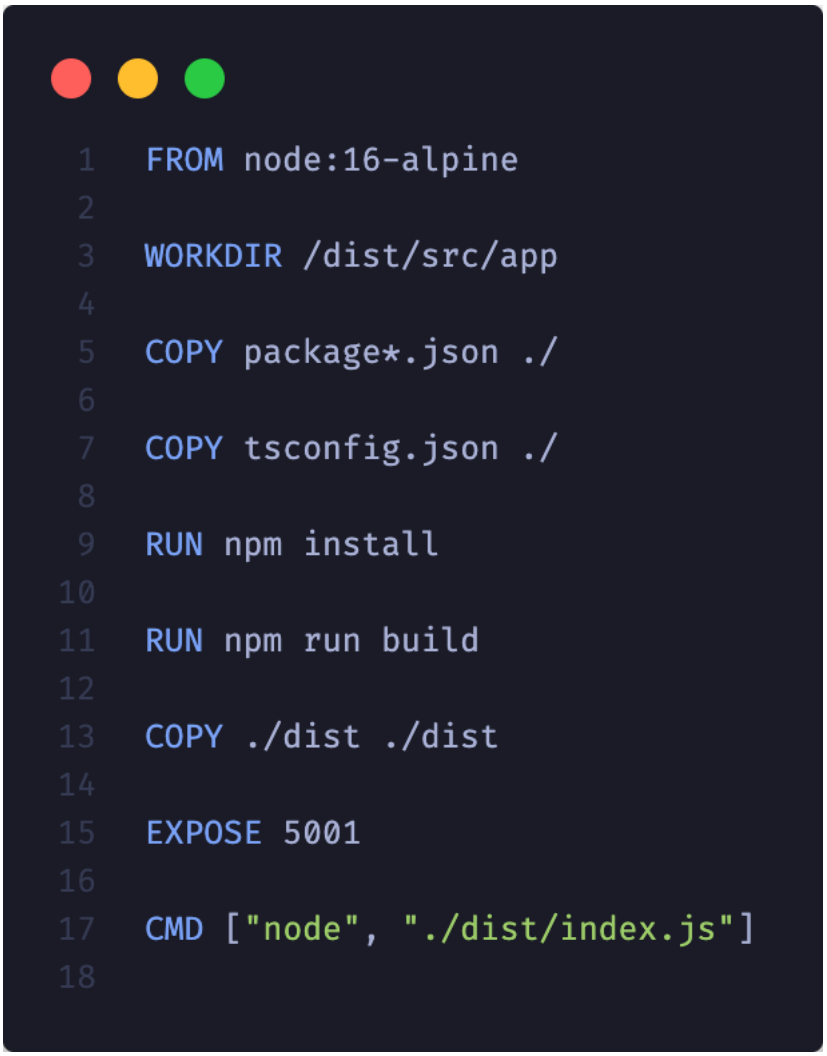
Responses

Code	Description	Links
200	Method for logging into the system Media type: application/json Controls Accept header. Example Value | Schema	*No links*

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6...",
  "role": "trabajador",
  "user": 1001
}
```

4.2.3.12. Dockerfile

A la hora de desplegar nuestra aplicación hacia los servidores de producción, la mejor manera es mediante la contenerización, con esa finalidad creamos un fichero *Dockerfile* el cual contiene las instrucciones para construir el contenedor de la aplicación.



```
1  FROM node:16-alpine
2
3  WORKDIR /dist/src/app
4
5  COPY package*.json ./
6
7  COPY tsconfig.json ./
8
9  RUN npm install
10
11 RUN npm run build
12
13 COPY ./dist ./dist
14
15 EXPOSE 5001
16
17 CMD ["node", "./dist/index.js"]
18
```

La captura de código presentada es el contenido del fichero *Dockerfile,* como podemos contemplar está formado por diferentes comandos que explicamos de seguida.

- **FROM node:16-alpine**: Mediante este comando indicamos la imagen base que se utiliza para construir la imagen *Docker* de nuestra aplicación. En este caso utilizamos la versión 16 de *Node* basada en *Alpine Linux*, esta es una distribución de *Linux* muy ligera y compacta que ayuda a ahorrar espacio.
- **WORKDIR**: Especificamos el directorio de trabajo dentro del contenedor, es decir indicamos donde se ejecutan los siguientes comandos (*COPY, RUN*, etc).
- **COPY**: Copiamos al directorio de trabajo los ficheros que empiezan con *package* y tienen la extensión *.json.* También copiamos el fichero de configuración de *TypeScript* (*tsconfig.json*).
- **RUN**: El comando *RUN* se utiliza para ejecutar comandos en el momento de creación de la imagen, en este caso, instalamos las dependencias y creamos el código compilado de la aplicación.

Nota: *TypeScript* es utilizado solamente para desarrollar la aplicación, a la hora de desplegarla, tenemos que pasar de código *TypeScript* a *JavaScript*. La funcionalidad del comando ***npm run build*** esta especificada en el fichero *package.json*.

```
"scripts": {
    "tsc": "tsc",
    "dev": "set debug=* && ts-node-dev --respawn --transpile-only ./src/index.ts",
    "build": "tsc && cp ormconfig.json dist/",
    "start": "ts-node-dev --respawn --transpile-only ./src/index.ts"
  }
```

- **COPY ./dist ./dist:** Copiamos el código compilado creado por el comando anterior a la carpeta */dist* del contenedor.
- **EXPOSE 5001:** Exponemos dentro del contenedor el puerto el cual utiliza la aplicación para resolver peticiones.
- **CMD:** El comando *CMD* a diferencia de *RUN*, ejecuta el comando una vez creado y ejecutándose el contenedor, en esta ocasión, ejecuta el que levanta la aplicación.

Para crear la imagen *Docker* de la aplicación basándose en nuestro fichero Dockerfile utilizamos este comando:

```
docker build -t <nombre_imagen>:<versión> .
```

4.2.3.13. *Logger*

Justo después de tener la aplicación corriendo en producción, nos puede interesar saber qué está pasando en cada instante debido a una incidencia o simplemente para mejorar el rendimiento de esta. Sin tener implementado un sistema que guarde un registro de *logs* (*Logger*) se complica la tarea.

```
const dotenv = require("dotenv");
dotenv.config();

import winston = require("winston");
const { ElasticsearchTransport } = require("winston-elasticsearch");

const logLevels = {
  error: 0,
  warn: 1,
  info: 2,
  http: 3,
  debug: 4,
};
const esTransportOpts = {
  level: "http",
  index: process.env.elasticsearch_index,
  node: process.env.elasticsearch_node,
  clientOpts: { node: process.env.elasticsearch_node },
};

const esTransport = new ElasticsearchTransport(esTransportOpts);
. . .

```

La primera solución pensada se trata de configurar el paquete *Winston* para enviar los *logs* a una base de datos de *Elasticsearch [28]* y utilizar *Grafana [29]* para mostrar estos datos en forma de gráficas, de seguida estudiamos como se ha implementado esta solución.

En el fichero *logger.ts* tenemos la configuración necesaria, en la captura anterior podemos ver que establecimos diferentes niveles de *logs* en la variable ***logLevels*** y en ***esTransportOpts*** indicamos a donde enviaremos los *logs*, en este caso a un *Elasticsearch*. En la última línea creamos una instancia del objeto ***ElasticsearchTransport***.

```
. . .
const logger = winston.createLogger({
  levels: logLevels,
  format: winston.format.combine(
    winston.format.timestamp({ format: "YYYY-MM-DD HH:mm:ss" }),
    winston.format.printf((info) => {
      const { timestamp, level, message, ...meta } = info;
      let logMessage = `${timestamp} [${level}]: ${message}`;

      if (Object.keys(meta).length > 0) {
        logMessage += "\n" + JSON.stringify(meta, null, 2);
      }

      return logMessage;
    })
  ),

  transports: [
    new winston.transports.File({ filename: "logs/error.log", level: "error" }),
    new winston.transports.File({ filename: "logs/warn.log", level: "warn" }),
    new winston.transports.File({ filename: "logs/info.log", level: "info" }),
    new winston.transports.File({ filename: "logs/http.log", level: "http" }),
    new winston.transports.File({ filename: "logs/debug.log", level: "debug" }),
    esTransport,
  ],
});
```

La captura presentada es lo que resta del fichero *logger.ts*, donde vemos cómo se crea la instancia de *winston* pasándole la configuración en forma de objeto. También se puede notar como es el formato de los *logs*, en este caso, se utiliza ***winston.format.combine()*** para combinar varios formatos. El formato incluye un sello de tiempo y una función personalizada para formatear los registros.

Una vez implementado *Winston* en nuestra aplicación, debemos proceder a crear la base de datos de *Elasticsearch* y conectarlo a *Grafana*. Para tener *Elasticsearch* y *Grafana* podemos instalarlos como contenedores de *Docker*.

```
docker network create obras-proyecto
```

Primero creamos una red para conectar entre los diferentes contenedores.

```
docker run --name mysql-container -d -p 3306:3306 --network obras-proyecto -e MYSQL_ROOT_PASSWORD=PASS
docker run --name elasticsearch -d -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:7.14.0
docker run --name grafana -d -p 3000:3000 grafana/grafana
docker run --name api-dashboard -d -p 5001:5001 --network obras-proyecto node-api-dashboard:1
```

Una vez tenemos creada la red (obras-proyecto) pasamos a instalar y ejecutar los contenedores *mysql-container, elasticsearch* y *grafana*.

- **d:** Ejecuta el contenedor en modo "*detach*" (en segundo plano).
- **--name:** Asigna un nombre al contenedor.
- **-p xxxx:xxxx:** Mapea los puertos del contenedor a los puertos del *host* para que se pueda acceder a este.
- **--network:** Especifica la red a la cual pertenecerá el contenedor.
- **-e:** Para pasar al contenedor las variables de entorno.

Como podemos notar, cada vez que queremos levantar o parar los contenedores lo tenemos que hacer uno por uno, lo que puede llegar a ser molesto, como arreglo, utilizamos *Docker-compose [30]*.

La siguiente captura (siguiente página) muestra como es el contenido de nuestro *Docker-compose*. Este contiene todas los comandos presentados en un solo fichero, de esta forma, con un solo comando tratamos los diferentes contenedores.
Para levantar o para los contenedores con *Docker-compose* es de la siguiente manera:

```
docker compose -f "docker-compose.yml" up -d
```

```
docker compose -f "docker-compose.yml" down
```

```yaml
version: '3'
services:
  node-api-dashboard:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "5001:5001"
    networks:
      - obras-proyecto

  mysql-container:
    image: mysql
    container_name: mysql-container
    ports:
      - "3306:3306"
    networks:
      - obras-proyecto
    environment:
      MYSQL_ROOT_PASSWORD: PASS

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.14.0
    container_name: elasticsearch
    ports:
      - "9200:9200"
      - "9300:9300"
    networks:
      - obras-proyecto
    environment:
      discovery.type: single-node

  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - "3000:3000"

networks:
  obras-proyecto:

```

Al tener todo el entorno creado y los contenedores conectados, ya podemos continuar con la implementación del sistema *Logger*. El siguiente paso a realizar, es crear el índice que alojará los datos de los *logs*.

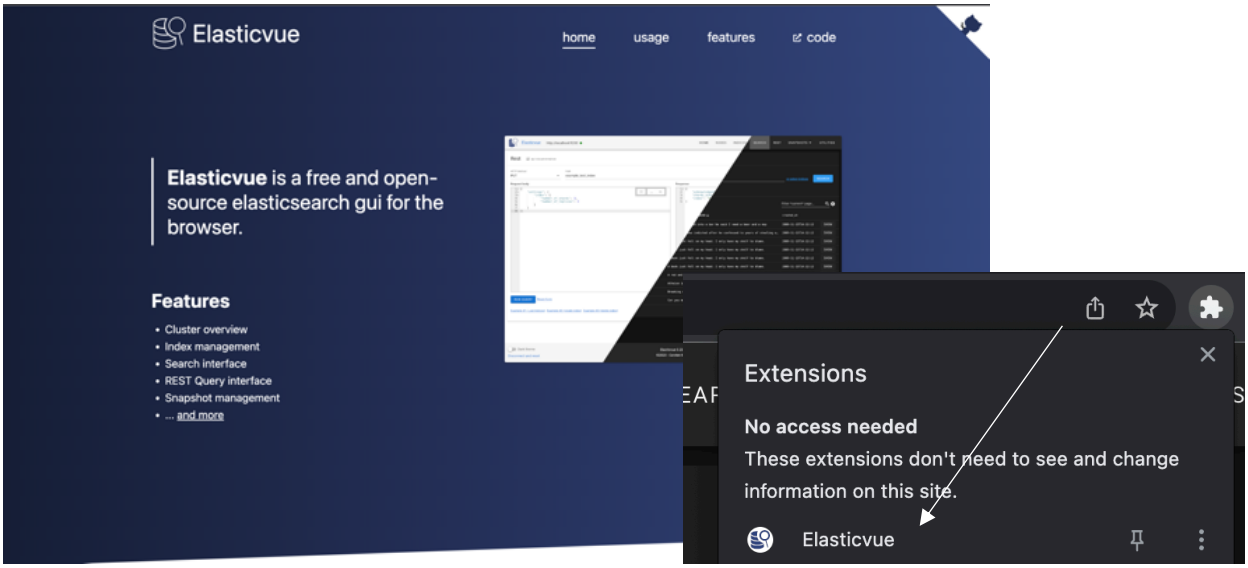
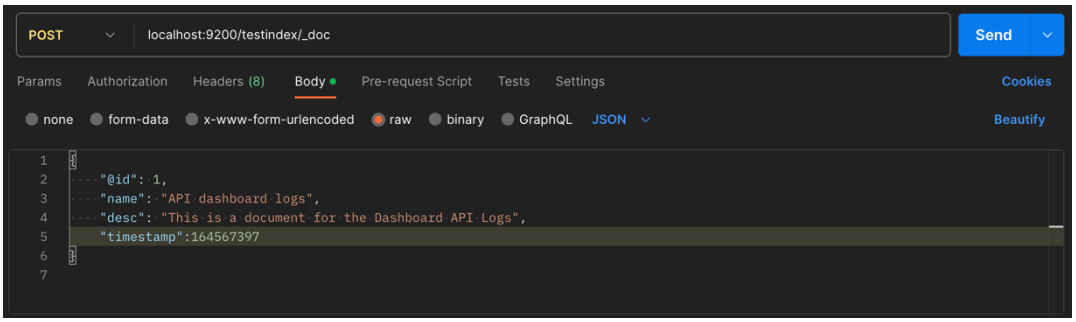
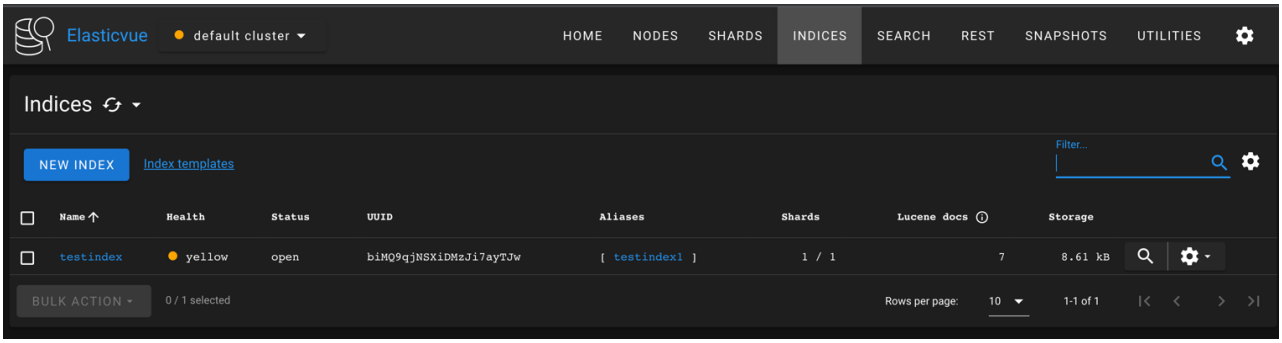


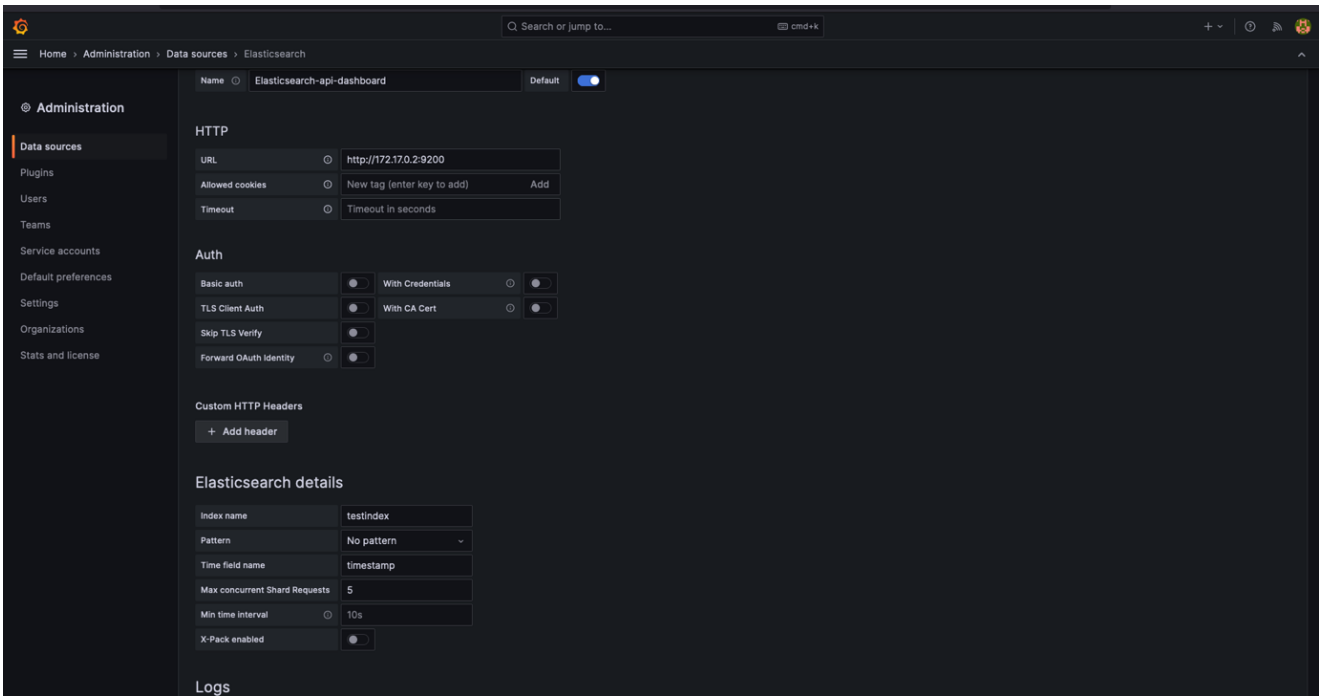
Figura X. Captura de pantalla de la extensión *Elasticvue*.
https://github.com/cars10/elasticvue

Para visualizar los datos de *Elasticsearch*, utilizamos la extensión ***Elasticvue*** que se instala en nuestro navegador de manera fácil y rápida.
Mediante *Postman [31]* creamos un índice dentro de *Elasticsearch* para guardar nuestros registros.

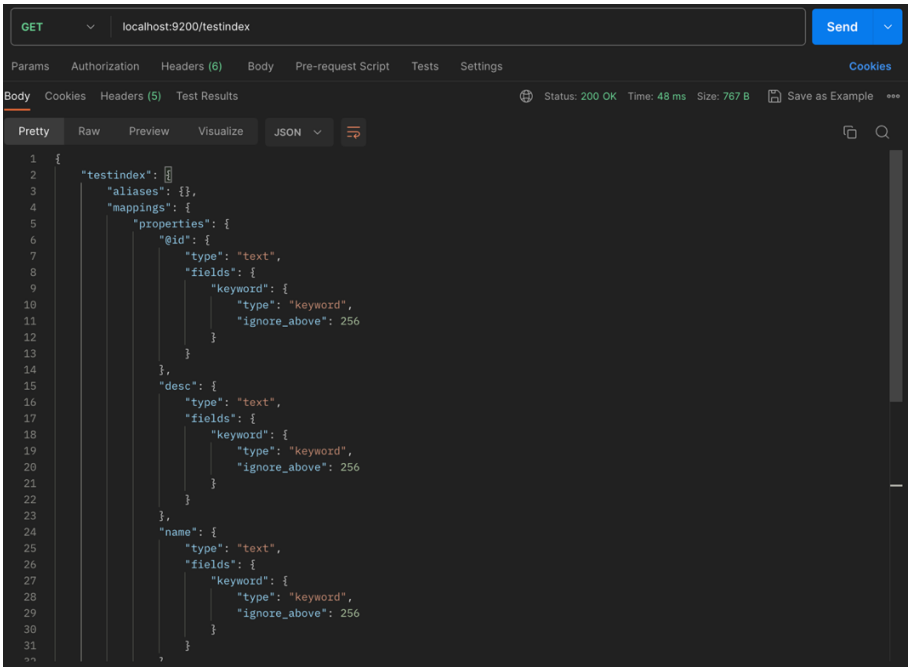
Inmediatamente después de crear el índice, accedemos a nuestro contenedor de *Grafana* vía navegador y procedemos a conectarle la fuente de datos (Elasticsearch).



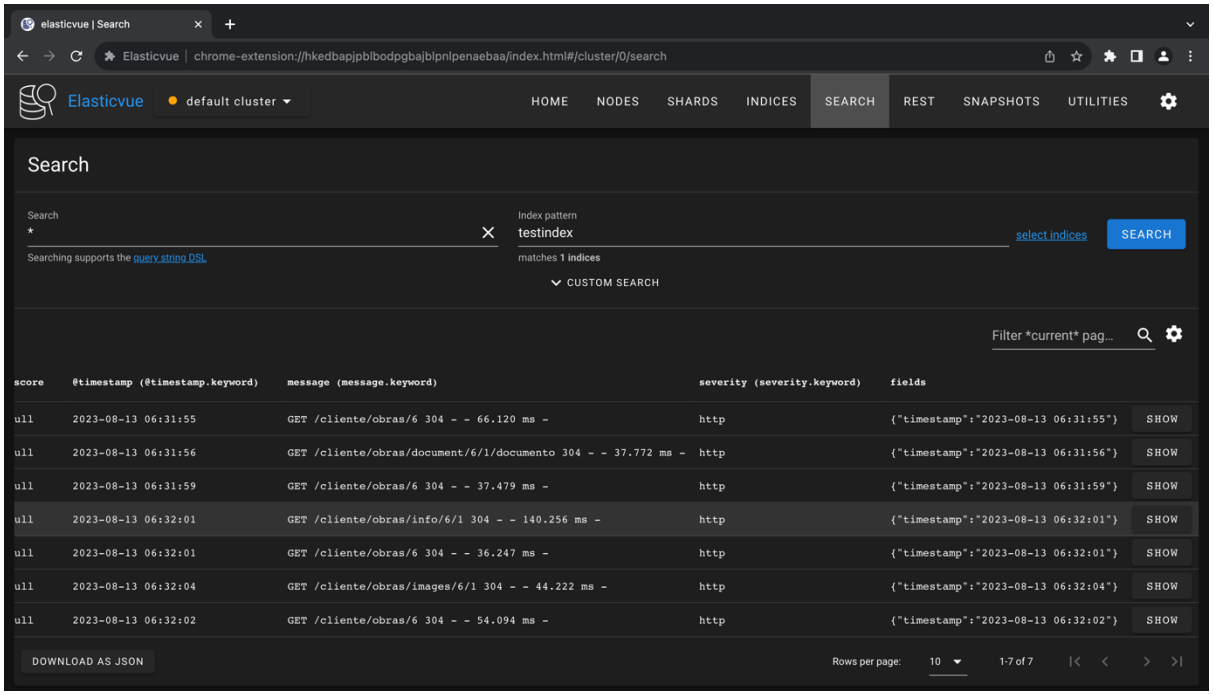


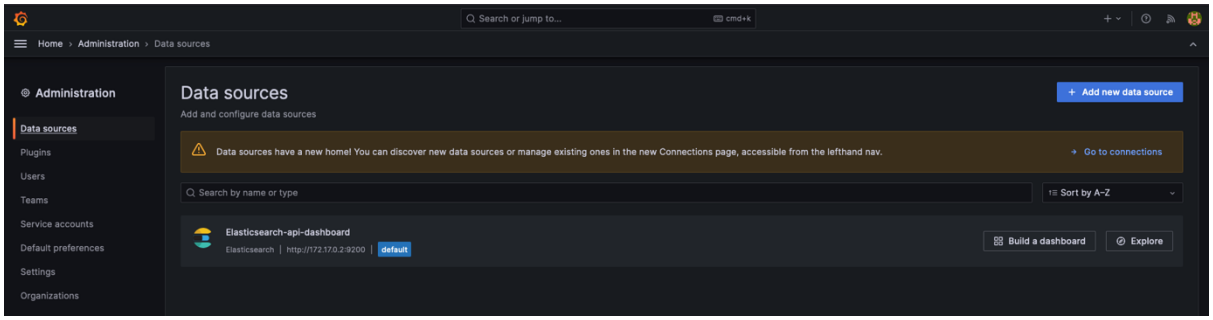


Nota: en el mismo proceso de configuración de *Grafana*, obtenemos un error (*NO DateTimeield ts found*) el cual se resuelve de la siguiente manera:



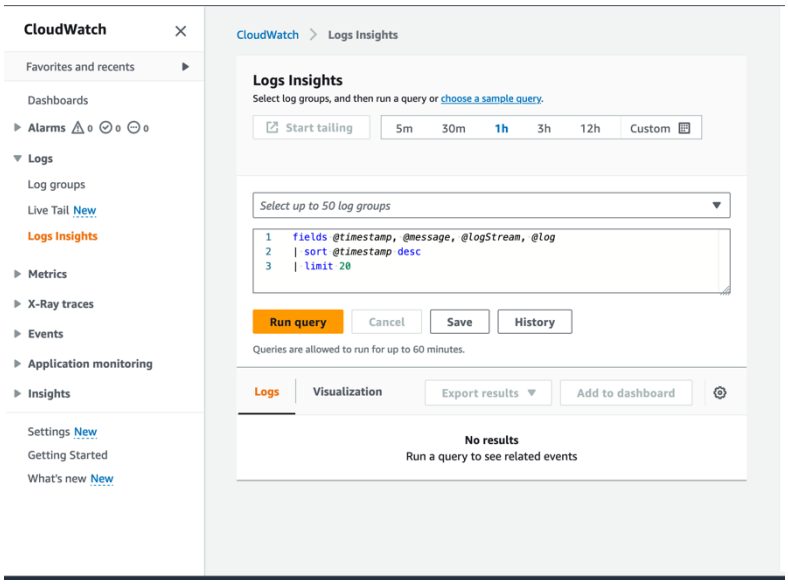
Debemos recuperar los metadatos del índice creado y realizar una pequeña modificación en el campo *timestamp*. Copiamos los metadatos obtenidos en la figura de la izquierda, realizamos la modificación en ***timestamp*** y guardamos los cambios (figura de la derecha).





En esta captura, ya podemos ver que los *logs* se están guardando de manera correcta en la base de datos de *Elasticsearch* y *Grafana* ya tiene asociado el mismo *dataset*. Por lo tanto, ya podemos crear los paneles para monitorizar los datos.

La otra alternativa es usar el servicio de *Amazon CloudWatch,* ya que, al desplegar nuestras aplicaciones en la plataforma de *AWS*, tendremos acceso tanto a *logs* generales como propios de cada contenedor.



4.3. Implementación back-end sistema presupuestación

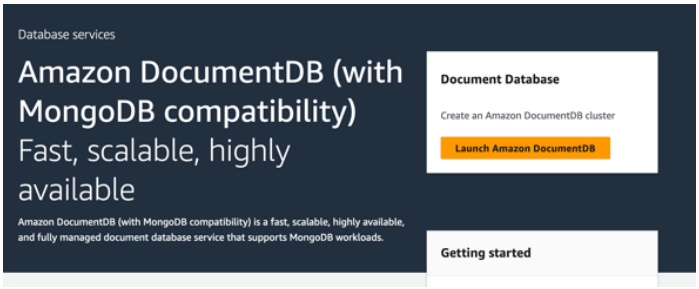
Para evitar la repetición de contenido, en el back-end del sistema de presupuestación explicamos aquellos conceptos que consideramos importantes y diferentes a la *API* anterior. Cabe señalar que también dispone de la mayoría de las funcionalidades ya explicadas como es el sistema de rutas, *logger*, *testing* y contenerización y más.

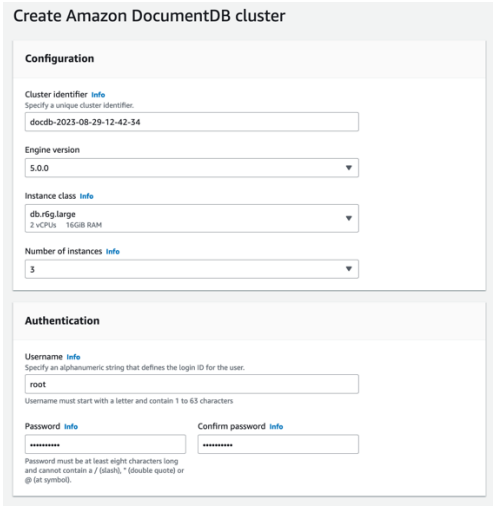
La primera diferencia que podemos comentar es el uso de una base de datos no relacional, ya que como podemos notar en el punto [*3.2.3.1. Diseño del sistema de presupuestación*], las tablas diseñadas no mantienen una relación entre sí, básicamente se trata de tablas que contienen una serie de precios que podemos consultar. Debido a esto, hemos optado por utilizar un servicio de *AWS* compatible con la famosa base de datos *NoSQL mongoDB* (*Amazon DocumentDB)*

4.3.1. Amazon DocumentDB

Amazon DocumentDB [32] ha sido diseñado como una solución de alto rendimiento, escalable y altamente disponible. Lo más importante es su compatibilidad con *mongoDB*, lo cual permite aprovechar los conocimientos ya adquiridos con este tipo de bases de datos.

Primeramente, debemos crear un *clúster* de la siguiente manera:





Como podemos ver, a la hora de crear el *cluster*, debemos indicar una serie de configuración que son:

- El usuario *Root*
- La contraseña del usuario *Root*
- El puerto para acceder a la base de datos.
- La frecuencia para realizar las copias de seguridad de los datos (*Backup*).
- El número de instancias o replicas que tendremos.
- Entre otras cosas.

Llegados a este punto, mediante aplicaciones como ***MongoDB Compass***, podemos conectarnos a la base de datos y crear los esquemas que contendrán los precios a consultar.

4.3.2. Datos del Presupuestador

La aplicación front-end después de proporcionar un presupuesto a un cliente es muy importante guardar un registro de este hecho, ya que a la empresa le interesa tener en su poder los datos del cliente, las medidas introducidas y sobre todo el precio obtenido por la reforma. Por esta razón hemos creado unas tablas donde guardar estos datos. Las tablas pertenecen a una base de datos *MySQL*.
Como se trata de un conjunto de tablas que se van a crear una sola vez, esta vez hemos empleado el lenguaje *SQL* para su creación.

```sql
CREATE TABLE
    presupuestador_web_user (
        id VARCHAR(200) NOT NULL PRIMARY KEY,
        nombre VARCHAR(200) NOT NULL,
        telefono VARCHAR(50) NOT NULL,
        email VARCHAR(200) NOT NULL,
        codigo_postal VARCHAR(50) NOT NULL
    ) ENGINE = InnoDB;

CREATE TABLE
    presupuestador_web_user_precios (
        id VARCHAR(200) NOT NULL PRIMARY KEY,
        total VARCHAR(200),
        cocina VARCHAR(200),
        bano VARCHAR(200),
        aseo VARCHAR(200),
        pavimento_gres VARCHAR(200),
        pavimento_laminado VARCHAR(200),
        falso_techo VARCHAR(200),
        pintura VARCHAR(200),
        madera VARCHAR(200),
        aluminio VARCHAR(200),
        electricidad VARCHAR(200),
        fontaneria VARCHAR(200),
        saneamiento VARCHAR(200),
        antena VARCHAR(200),
        calefaccion VARCHAR(200),
        aire_acondicionado VARCHAR(200),
        varios VARCHAR(200),
        pdf BLOB
    ) ENGINE = InnoDB;
```

```sql
CREATE TABLE
    presupuestador_web_user_medidas (
        id VARCHAR(200) NOT NULL PRIMARY KEY,
        superficie_vivienda DECIMAL(13, 2),
        altura_vivienda DECIMAL(13, 2),
        largo_cocina DECIMAL(13, 2),
        ancho_cocina DECIMAL(13, 2),
        largo_bano DECIMAL(13, 2),
        ancho_bano DECIMAL(13, 2),
        largo_aseo DECIMAL(13, 2),
        ancho_aseo DECIMAL(13, 2),
        num_puertas DECIMAL(13, 2),
        entradas_electricidad DECIMAL(13, 2),
        salones_electricidad DECIMAL(13, 2),
        cocinas_electricidad DECIMAL(13, 2),
        banos_electricidad DECIMAL(13, 2),
        aseos_electricidad DECIMAL(13, 2),
        habitaciones_electricidad DECIMAL(13, 2),
        pasillos_electricidad DECIMAL(13, 2),
        lavaderos_electricidad DECIMAL(13, 2),
        cocinas_fontaneria DECIMAL(13, 2),
        banos_fontaneria DECIMAL(13, 2),
        aseos_fontaneria DECIMAL(13, 2),
        lavaderos_fontaneria DECIMAL(13, 2),
        cocinas_saneamiento DECIMAL(13, 2),
        banos_saneamiento DECIMAL(13, 2),
        aseos_saneamiento DECIMAL(13, 2),
        lavaderos_saneamiento DECIMAL(13, 2),
        salones_antena DECIMAL(13, 2),
        cocinas_antena DECIMAL(13, 2),
        habitaciones_antena DECIMAL(13, 2),
        entradas_calefaccion DECIMAL(13, 2),
        salones_calefaccion DECIMAL(13, 2),
        cocinas_calefaccion DECIMAL(13, 2),
        banos_calefaccion DECIMAL(13, 2),
        aseos_calefaccion DECIMAL(13, 2),
        habitaciones_calefaccion DECIMAL(13, 2),
        pasillos_calefaccion DECIMAL(13, 2),
        lavaderos_calefaccion DECIMAL(13, 2),
        salones_aire DECIMAL(13, 2),
        cocinas_aire DECIMAL(13, 2),
        habitaciones_aire DECIMAL(13, 2)
    ) ENGINE = InnoDB;
```

En las dos capturas mostradas, podemos observar como creamos tres tablas utilizando el lenguaje *SQL*.
Estas tablas, como hemos mencionado previamente, se encargarán de guardar cualquier dato introducido o generado vía web.

4.3.3. Almacenamiento de datos del Presupuestador

Para poder almacenar los valores comentados en el apartado anterior, debemos crear unas rutas, mediante las cuales conectaremos entre el fornt-end y el back-end.

```
router.route("/email-integral").post(email_cliente);
```

Una vez tenemos un *endpoint* que apunta a una función del *controller,* cuando se invoca esta ruta, procederá a ejecutar la función *email_cliente*.

```
export async function email_cliente(req: Request, res: Response) {
  await pdfEmailCliente(req, res);
  await crearUsuarioPresupustoWeb(req, res);
}
```

El método *email_cliente* internamente contiene la llamada a otras dos funciones:

- *pdfEmailCliente*: Esta función se encarga de generar un documento tipo PDF, el cual contiene los datos de contacto del cliente que ha solicitado el presupuesto, las medidas proporcionadas y el coste de reforma de cada sesión.
  Debido a la exigencia de la empresa por ofrecer PDF modernos y con un diseño agradable, hemos optado por generar el documento PDF a partir del código *HTML* con estilos *CSS* (siguiente apartado).
  Una vez se ha generado el documento y guardado localmente en el sistema de archivos de la aplicación, se procede a enviar un correo electrónico al cliente adjuntando el documento recién creado (lo veremos más adelante).
- *crearUsuarioPresupuestoWeb*: Este método se encarga de guardar los datos proporcionados por la web en nuestras tablas, en concreto, almacena los datos de contacto del cliente en la tabla *presupuestador_web_user*, el presupuesto de cada parte en *presupuestador_web_user_precios* y finalmente, las medidas empleadas en el cálculo en *presupuestador_web_user_medidas*.

```
const conn = await connect();
await conn.query("INSERT INTO presupuestador_web_user SET ?", newWebUser);
await conn.query(
  "INSERT INTO presupuestador_web_user_precios SET ?",
  newWebUserReforma
);
await conn.query(
  "INSERT INTO presupuestador_web_user_medidas SET ?",
  newWebUserMedidas
);
```

```
export async function connect() {
  const connection = await createPool({
    host: "localhost",
    port: 3306,
    user: "****",
    password: "****",
    database: "tfg_db_presupuestador",
    multipleStatements: true,
    connectionLimit: 5000,
  });
  return connection;
}
```

```
let newWebUser = {
  id: User.id,
  nombre: User.nombre,
  telefono: User.telefono,
  email: User.email,
  codigo_postal: User.codigo_postal,
};
```

Un cliente después de obtener el presupuesto de su reforma en la web, si desea obtener un documento con más destalles debe introducir sus datos de contacto y el correo donde desea recibir este documento. Como hemos mencionado anteriormente, con el objetivo de crear un documento moderno, utilizamos código *HTML* con estilos *CSS*.

```
const email_html_page =
  `
  <!DOCTYPE html>
      <html>
          <head>
              <meta name="viewport" content="width=device-width, initial-scale=1">
              <meta charset="utf-8">
          </head>
          <body>
              <div  style="padding: 10px; background: #fff; max-width: 560px;    border-radius: 20px;margin: 60px auto;box-shadow: 0 7px 30px -10px rgba(150,170,180,0.5);">
                  <h4>Hola ` +
                      User.nombre +
                  `</h4>
                  <p>Te agradecemos que utilices nuestros servicios y confiar en nosotros, <span style="color: #fb6919 ;">Formaremos un buen equipo!</span></p>
                  <button style="background-color:  #fb6919 ;
                                  border: none;color: white;
                                  padding: 15px 32px;
                                  text-align: center;
                                  text-decoration: none;
                                  display: inline-block;
                                  font-size: 16px;margin: 4px 2px;
                                  cursor: pointer; ">

                      <a href="http://www.REFORMA.net/" style="text-decoration: none;
                                                               color:white;">
                          Ir a la página
                      </a>
                  </button>
                  <br/><br/>
              </div>
          </body>
      </html>
  `;
```

En esta captura podemos observar el código *HTML* implementado para poder dar estilo al correo que va a recibir nuestro cliente. La misma idea es empleada para crear el documento que será adjuntado al correo.

```
await pdf
    .create(html_pdf_todo)
    .toFile("src/public/" + User.id + ".pdf", function (err: any, res: any) {
      if (!err) {
        sleep(2000).then(async () => {
          // Body
        });
```

Una vez guardado el código *HTML* en la variable *html_pdf_todo,* mediante la librería *html-pdf* creamos el nuevo documento tomando como fuente el código *HTML*. El fichero creado será guardado en la carpeta pública del proyecto (*src/public*). Para evitar sobrescribir archivos con el mismo nombre, utilizamos como nombre el identificador del usuario.

4.3.5. Envío de correo

Una vez creado el documento, procedemos a enviar el correo adjuntando dicho fichero.

```
try {
  const params = {
    EmailAddress: client.email,
  };
  ses.verifyEmailAddress(params, async (err: any, data: any) => {
    if (err) {
      console.error(
        "Error al verificar la dirección de correo electrónico:",
        err
      );
    } else {
      // Email verifyed
    }
  });
} catch (err) {
  console.error(err);
}
```

Antes de enviar al cliente el correo con el documento, primero debe verificar su cuenta, para ello, enviamos un correo con un enlace de verificación mediante la función *verifyEmailAddress* que nos facilita el servicio *SES* de *AWS*.

```
if (fs.existsSync(loc_pdf)) {
  const mailOptions = {
    from: process.env.email,
    to: client.email,
    subject: "Correo desde el sistema de presupuestación.",
    html: email_html_page,
    attachments: [
      {
        filename: fileName,
        path: loc_pdf,
        contentType: "application/pdf",
      },
    ],
  };

  try {
    const info = await transporter.sendMail(mailOptions);
    return res.status(code).jsonp("Correo enviado correctament!");

  } catch (error) {
    return res.status(code).jsonp("Error al enviar el correo!");
  }
}
```

Finalmente, procedemos al envío del correo electrónico hacia la cuenta del cliente adjuntando el documento PDF creado.

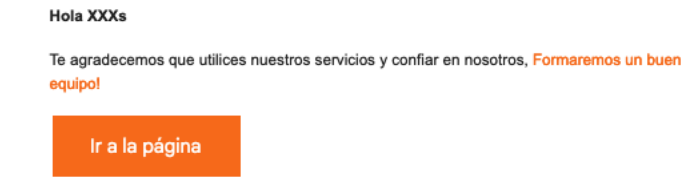


***Figura 35**. Correo recibido por el cliente.*

En la Figura 35 vemos un ejemplo del correo electrónico que recibe el cliente después de solicitar su presupuesto por la web, este correo incluye el PDF detallado con los costes de cada parte.

En este ejemplo, el documento tiene una sola página debido a que el cliente ha seleccionado reformar tan solo cuatro partes, de lo contrario contendrá más páginas.

4.4. Implementación front-end aplicación administrativa

Al terminar la explicación del back-end de la aplicación administrador, será momento de continuar con la explicación, esta vez con la parte del front-end de esta misma.
Como hemos visto en los diagramas *UML*, la aplicación tienes diferentes funcionalidades dependiendo del rol del usuario.

4.4.1. Componentes fundamentales de Angular [33][34]

Antes de comenzar a revelar detalles y funciones sobre la aplicación, debemos hacer una pequeña introducción sobre aquellos componentes fundamentales que el *framework* Angular nos ofrece para desarrollar aplicaciones.

1. **Módulos (Modules):**
   Los módulos en Angular son bloques de construcción que ayudan a agrupar y a organizar la aplicación en componentes reutilizables, en otras palabras, permite dividir la aplicación en grupos de componentes y estos se pueden importar, lo que permite trabajar de forma modular. En la raíz de la aplicación disponemos de un módulo principal llamado *app.module.ts* y dentro de cada módulo creado habrá un módulo propio, el cual se tendrá que importar en el principal para poder ser utilizado.

2. **Componentes (Components):**
   Los componentes son una pequeña parte visual y funcional de la aplicación. Un componente en Angular representa una vista, esta puede tener tanto código *HTML*, *CSS* como lógica agrupada debajo de un directorio.

3. **Servicios (Services):**
   Los servicios con clases que contienen cierta lógica de negocio o funcionalidad a compartir entre componentes u otros servicios. Un servicio se utiliza muy a menudo para realizar peticiones *HTTP* hacia un servidor back-end.

4. **Directivas (Directives):**
   Las directivas son instrucciones que se aplican directamente sobre el *DOM*, e indican a Angular como debe comportarse. Existen dos tipos: directivas de atributo (cambian el comportamiento de un elemento) y directivas estructurales *(*ngIf y *ngFor*).

5. **Pipes:**
   Los pipes son muy útiles a la hora de realizar una transformación visual sobre un dato antes de mostrarlo en la interfaz de usuario.

6. **Enlaces de datos (*Data Binding*):**
   Como hemos mencionado antes, un componente en Angular esta agrupado en un directorio, este entre otros ficheros, tiene un fichero *HTML* y otro fichero de lógica, por esta razón nos puede interesar enlazar entre la variable del *HTML* y el fichero de lógica, para ello Angular entre otras alternativas, nos ofrece:
   - Enlace unidireccional: Mediante ***{{}}***.
   - Enlace de propiedad: Utilizando ***[property***].
   - Enlace de evento: con ***(event).***
   - Enlace bidireccional: Finalmente, se utiliza ***[(ngModel)].***

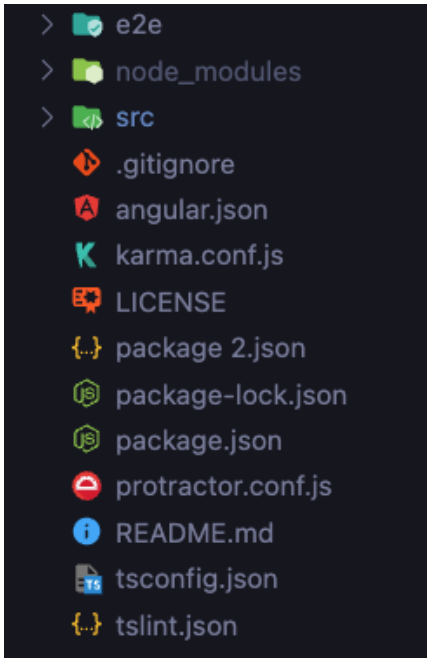
7. **Enrutamiento (*Routing*):**
   El enrutamiento nos permite navegar entre los componentes de una aplicación. Para poder utilizar el enrutamiento en una aplicación Angular, debemos importar el módulo *RouteModule* que viene instalado por defecto.

8. **Inyección de Dependencias (*Dependency Injection*):**
   Es muy importante entender el concepto de inyección de dependencias, ya que es la manera de proporcionar la instancia de los servicios a los componentes, dicho de otra forma, para que un componente pueda utilizar un servicio, tenemos que pasarle la instancia del servicio por su constructor, pero en ningún momento hacemos un *new*, porque se encarga el propio Angular de distribuir dicha instancia.

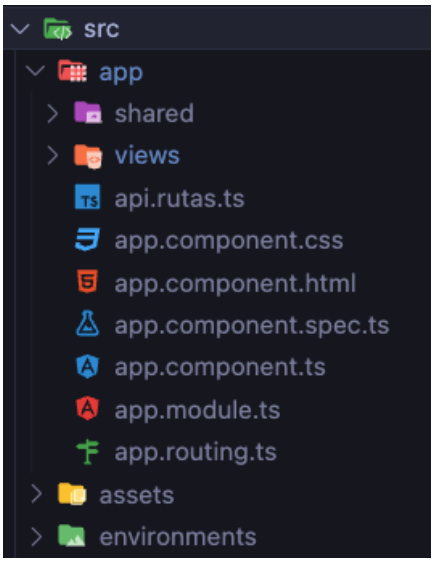
9. **Módulos de Formularios (*Forms Modules*):**
   Finalmente, Angular proporciona un módulo para manejar formularios *HTML* de manera profesional y controlada, de esta forma podemos crear formularios reactivos.

4.4.2. Estructura del proyecto

En la captura presentada, podemos ver aquellas carpetas que forman la aplicación web.



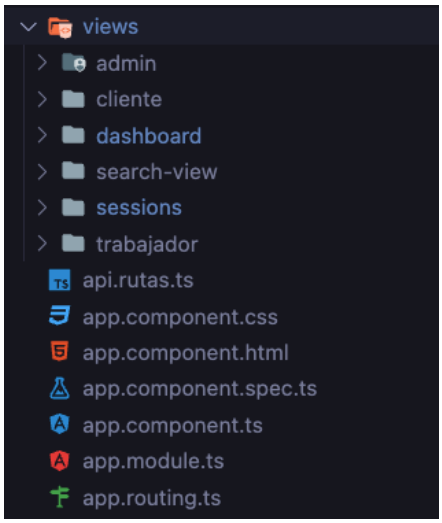
- **e2e**: La carpeta *e2e* contiene los tests *End-To-End* de los componentes.
- **node_modules**: Contiene las librerías necesarias utilizadas por los diferentes componentes.
- **src:** Abarca todo el código fuente que hemos ido codificando.
- **angular.json:** Este fichero se utiliza por la aplicación Angular para definir cómo se compila, construye y se ejecuta la aplicación.
- **Karma.conf.js:** Se trata de un archivo de configuración para personalizar las pruebas unitarias que se ejecutan utilizando la herramienta de *testing Karma* [35].
- **package.json:** Es el archivo de configuración que contiene información de la aplicación, sobre sus dependencias, metadatos y script a ejecutar.
- **Tsconfig.josn:** Al igual que el back-end, es el archivo de configuración de *TypeScript* para definir el comportamiento del compilador de nuestro proyecto.
- **tslint.json:** *Tslint* [36] es una herramienta (*slint* para *TypeScript*) que ayuda a mantener la calidad del código mediante una serie de reglas y patrones de codificación, de esta manera evitamos cambios innecesarios en nuestro repositorio a la hora de compartir el código.



Dentro del directorio *src* nos encontramos con las siguientes carpetas:

- **shared**: El directorio *shared* contiene todos los componentes que son compartidos en toda la aplicación, como: pipes, guards, models…etc
- **views**: En este directorio disponemos de los módulos propios, más adelante veremos qué es exactamente.
- **assets**: En el directorio *assets* ponemos todos los objetos estáticos, como imágenes o logos.
- **environment**: Environment contiene los diferentes ficheros de las variables de entorno.

4.4.3. Ficheros globales



Como podemos notar en la captura dada, dividimos la aplicación de los diferentes módulos, donde cada uno cumple una funcionalidad. En el directorio *views* tenemos el módulo del usuario *admin*, *cliente*, *trabajador* y el de *sessions* (contiene los componentes relacionados con el inicio de sesión).
También disponemos del componente *app* que viene a ser el primer componente que se crea y del cual cuelgan los demás.
Asimismo, disponemos de un módulo principal (*app.module.ts*) y un módulo de rutas (*app.routing.ts*)

```
@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    SharedModule,
    AdminModule,
    ClienteModule,
    TrabajadorModule,
    HttpClientModule,
    PerfectScrollbarModule,
    ToastrModule.forRoot(),
    RouterModule.forRoot(rootRouterConfig, { useHash: false }),
  ],
  declarations: [AppComponent],
```

+

```
  providers: [
    { provide: JWT_OPTIONS, useValue: JWT_OPTIONS },
    JwtHelperService,
    {
      provide: HTTP_INTERCEPTORS,
      useClass: TokemInterceptorService,
      multi: true,
    },
    { provide: ErrorHandler, useClass: ErrorHandlerService },
    { provide: HAMMER_GESTURE_CONFIG, useClass: GestureConfig },
    {
      provide: PERFECT_SCROLLBAR_CONFIG,
      useValue: DEFAULT_PERFECT_SCROLLBAR_CONFIG,
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: TokenInterceptor,
      multi: true,
    },
    { provide: ConfirmationService },
    { provide: MessageService },
  ],
  bootstrap: [AppComponent],
  entryComponents: [AddClienteComponent],
})
export class AppModule {}
```

Para poder utilizar los diferentes módulos en nuestra aplicación, estos se tienen que importar y definir dentro del módulo principal (*app.module*). También importamos los servicios a usar en la sección de *providers*.

Ocurre lo mismo con las rutas, pero lo trataremos en otro apartado (*rutas*) donde entraremos más en detalle.

4.4.4. Funcionalidad

En la actual sección detallamos aquellas funciones más destacadas del front-end de la aplicación administrativa.

4.4.4.1. Auth

La primera funcionalidad que vamos a explicar es aquella que hay detrás de sistema de autenticación de la aplicación.

El primer componente que mostramos a los usuarios antes de acceder a la aplicación es el de inicio de sesión, a fin de acceder a los recursos disponibles, el usuario tendrá que autenticarse utilizando sus credenciales. Dependiendo de la respuesta del back-end, la aplicación redirigirá al usuario a unas rutas u otras.

signin.componenets.ts:

```
@Component({
  selector: "app-signin",
  templateUrl: "./signin.component.html",
  styleUrls: ["./signin.component.scss"],
  animations: matxAnimations,
})
export class SigninComponent implements OnInit, AfterViewInit {
  signinForm: FormGroup;
  errorMsg = "";
  return: string;
  loading: Boolean;
```

Como podemos notar, en el componente de lógica, tenemos declarado un conjunto de variables para detectar los errores, si el componente se ha cargado y, sobre todo, con *FormGroup* creamos un formulario para obtener las credenciales que el usuario ha introducido en el formulario *HTML*.

```
constructor(
  private jwtAuth: JwtAuthService,
  private matxLoader: AppLoaderService,
  private router: Router,
  private toastr: ToastrService,
  private ls: LocalStoreService
) {}
```

La manera de utilizar un servicio es inyectándolo en el constructor del componente, en la captura presentada, podemos observar que inyectamos diferentes servicios, tanto propios como de librerías.

```
ngOnInit() {
  this.signinForm = new FormGroup({
    username: new FormControl("", Validators.required),
    password: new FormControl("", Validators.required),
    rememberMe: new FormControl(true),
  });
}
```

El primer método de un componente que se ejecuta después del constructor es el *ngOnInit,* de esta forma, una vez creado el componente, procedemos a crear el formulario que guardará las credenciales del usuario. Podemos ver también que añadimos la validación necesaria para cada campo.

En el código *HTML*, hacemos referencia al formulario creado en el otro fichero (signin.component.ts) con un enlace bidireccional *[formGroup]*.
También le indicamos que método a debe invocar cuando se envíe el formulario *signin()*.

```
<form
        #loginForm="ngForm"
        [formGroup]="signinForm"
        class="signup4-form grey-100"
        (ngSubmit)="signin()"
      >
```

```
<div *ngIf="loginForm.form.valid">
  <button-loading
    [loading]="loading"
    loadingText="Iniciando sesión ... "
    class="mr-16"
    color="#F56600"
    >Entrar</button-loading
  >
</div>
```

Como podemos notar, el botón para enviar el formulario está englobado dentro de un *div* que contiene una condición. Este contenedor será visible solamente cuando los requisitos del formulario se hayan cumplido, es decir, que todas las validaciones establecidas en el formulario se tienen que cumplir.

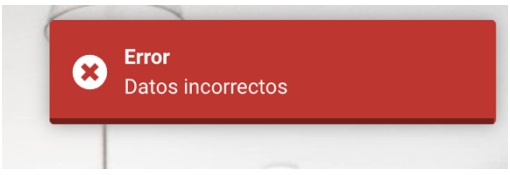
```
signin() {
  const signinData = this.signinForm.value;
  this.loading = true;
  this.jwtAuth.signin(signinData.username, signinData.password).subscribe(
    (response: any) => {
      this.onSucess("Bienvenido");

      this.ls.setItem("token", response.token);
      this.ls.setItem("role", response.role);
      this.ls.setItem("user", Number(response.user));

      this.loading = false;
      this.router.navigate(["dashboard"]);
    },
    (err) => {
      this.onError("Datos incorrectos");
      this.loading = false;
    }
  );
}
```

El método para ejecutar (*onsubmit Event*) al enviar el formulario es *signin,* esta función realiza estas funciones:

- Primero recupera los datos que el usuario ha introducido en el formulario.
- Mostramos el *Progress spinner*.
- Invocamos la función *signin* del servicio *jwtAuth* pasandole el correo y contraseña introducidos. Esta función se conecta con el endpoint del back-end para obtener el *token*.
- Una vez se ha resuelto la promesa, si ha sido exitosa, procedemos a guardar los datos en el ***localstorage*** del navegador, notificar al usuario de la operación exitosa y finalmente, redirigimos al componente principal. En caso contrario, mostraremos un mensaje de error.

Para mostrar los mensajes flotantes, usamos la librería *ngx-toastr* implementando dos funciones, la correspondiente a un error y la otra, en caso de éxito. El resultado se puede ver en las dos capturas facilitadas.

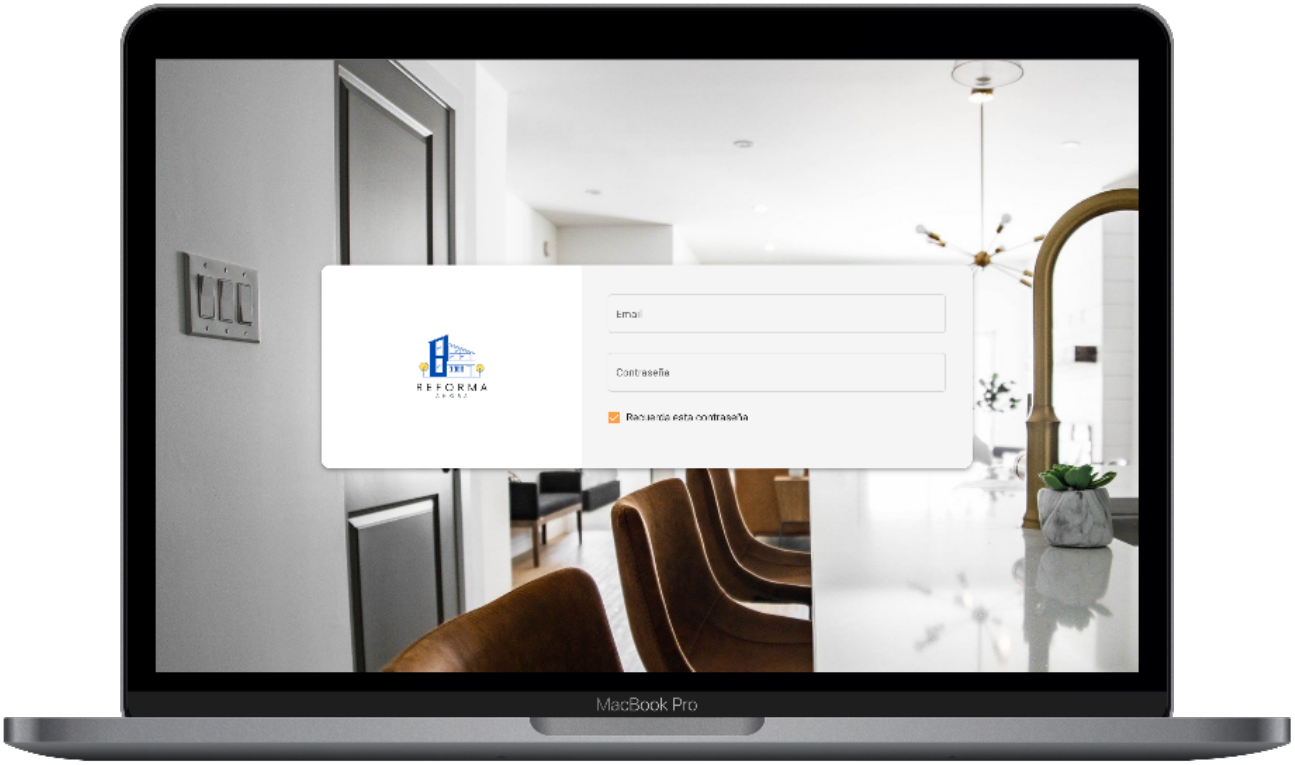
```
onSucess(message) {
  this.toastr.success(message, "Sesión");
}

onError(message) {
  this.toastr.error(message, "Error", {
    timeOut: 4000,
    progressBar: true,
    progressAnimation: "increasing",
  });
}
```



```
public signin(username, password) {
  let user = {
    username: username,
    password: password
  }
  return this.http.post(this.rutas.login, user)
}
```

El método llamado del servicio *jwtAuth.signin* realiza una llamada ***POST*** al Endpoint de nuestro back-end para comprobar las credenciales del usuario y obtener así el *token* correspondiente.
Para realizar llamadas *HTTP*, usamos el paquete *HttpClient* proporcionado por Angular.

signin.componenets.html:



4.4.4.2. *Rutas*

A medida que continuamos con el desarrollo de nuestra aplicación, esta irá creciendo y tendremos bastantes módulos y componentes que cargar, con el sistema de rutas tradicional, al ejecutar la aplicación, esta cargará todos los recursos en el mismo instante, como consecuencia, tendremos un rendimiento bajo y un tiempo de carga alto.
Para poder navegar entre los diferentes componentes de manera eficiente, debemos de implementar la alternativa proporcionada por el mismo equipo de Angular, el *Lazy loading* [37].

El *Lazy loading* es una técnica para mejorar la eficiencia y, sobre todo, el rendimiento de nuestra aplicación. La idea es cargar aquellos recursos necesarios en cada momento, en nuestro caso, cuando el usuario accede a una ruta, cargamos los componentes relacionados solamente. A continuación, vemos cómo hemos implementado esta estrategia.

```
. . .
{
  path: "",
  component: AdminLayoutComponent,
  canActivate: [AuthGuard, AdminGuard],
  children: [
    {
      path: "admin",
      loadChildren: () =>
        import("./views/admin/admin.module").then((m) => m.AdminModule),
      data: { title: "admin", breadcrumb: "Administrador" },
    },
  ],
},
. . .
```

La captura del código pertenece al fichero de rutas principal *app.routing.ts*, donde creamos las rutas principales, en este caso podemos observar que la ruta llamada */admin* apunta a otro módulo que contiene otro fichero de subrutas, de esta forma, dividimos la carga de la aplicación en módulos.

El contenido del *módulo /views/admin/admin.module* es el siguiente:

```
imports: [
  . . .
  RouterModule.forChild(AdminsRoutes),
],
exports: [
  . . .
],
providers: [{ provide: MAT_DATE_LOCALE, useValue: "en-GB" }],
})
export class AdminModule {}
```

Como podemos observar en la figura, el módulo contiene importaciones de otros módulos y componentes necesarios, pero debemos fijarnos en el fichero de rutas proporcionado. De esta forma, el módulo *AdminsModule* tiene constancia de las subrutas de */admin.*

```
export const AdminsRoutes: Routes = [
  {
    path: "",
    canActivate: [AuthGuard, AdminGuard],
    children: [
      {
        path: "cliente",
        component: ClienteComponent,
        data: { title: "Cliente", breadcrumb: "Cliente" },
      },
      {
        path: "documentos",
        component: DocumentosComponent,
        data: { title: "Documentos", breadcrumb: "Documentos" },
      },
      {
        path: "imagenes",
        component: ImagenesComponent,
        data: { title: "Imágenes", breadcrumb: "Imágenes" },
      },
      {
        path: "obra",
        component: ObraComponent,
        data: { title: "Obra", breadcrumb: "Obra" },
      },
      {
        path: "trabajador",
        component: TrabajadorComponent,
        data: { title: "Trabajador", breadcrumb: "Trabajador" },
      },

      /**Esta ruta deberia ir dentro de admin/documentos/upload-file */
      {
        path: "upload-file",
        component: UploadComponent,
        data: { title: "Subir Documentos", breadcrumb: "documentos" },
      },
      {
        path: "upload-image",
        component: UploadImageComponent,
        data: { title: "Subir Imágenes", breadcrumb: "imágenes" },
      },
      {
        path: "ver-pdf/:url",
        component: PdfViewerComponent,
        data: { title: "Ver PDF", breadcrumb: "ver-pdf" },
      },
    ],
  },
];
```

Las subrutas que cuelgan de la dirección */admin* son las que aparecen en la captura mostrada, donde cada una de estas subrutas está asociada a un componente, es decir, si por ejemplo el usuario accede a la ruta */admin/imagenes* se le presentará el componente *ImagesComponente*.

También es importante notar que en cada fichero de rutas asignamos un conjunto de *Guards*, tanto para verificar si el usuario está autenticado en el sistema como si dispone el rol adecuado.

El proceso mostrado de ejemplo se aplica a las demás rutas y los diferentes módulos disponibles.

Una vez creadas las rutas que los usuarios pueden utilizar, procedemos a mostrarlas dentro de la aplicación. Para esto:

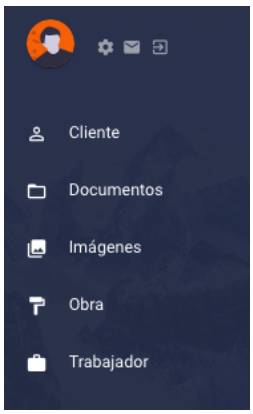
```typescript
@Injectable()
export class NavigationService {
  constructor() {}
  iconMenu: IMenuItem[] = [
    //Admin
    {
      name: "Adminisrador",
      type: "separator",
    },
    {
      name: "Cliente",
      type: "link",
      tooltip: "Administrador",
      icon: "perm_identity",
      state: "admin/cliente",
    },
    . . .

    //Cliente
    {
      name: "Cliente",
      type: "separator",
    },
    {
      name: "Documentos",
      type: "link",
      tooltip: "Cliente",
      icon: "folder_open",
      state: "cliente/documentos",
    },
    . . .

    // Trabajador
    {
      name: "Trabajador",
      type: "separator",
    },
    {
      name: "Obra",
      type: "link",
      tooltip: "Trabajador",
      icon: "folder_open",
      state: "trabajador/obra",
    },
    . . .
  ];

  iconTypeMenuTitle: string = "Frequently Accessed";
  menuItems = new BehaviorSubject<IMenuItem[]>(this.iconMenu);
  menuItems$ = this.menuItems.asObservable();

  publishNavigationChange(menuType: string) {
    this.menuItems.next(this.iconMenu);
  }
}
```

En el servicio llamado *navigation.service.ts* tenemos declarado los enlaces de los diferentes usuarios. Como podemos notar, los enlaces tienen un campo llamado *tooltip*, este campo es utilizado para filtrar los enlaces por diferentes roles posteriormente.
Este servicio es importado en el componente *sidebar.component.ts*, el cual se encarga de mostrar las rutas mediante el elemento *sidebar*.

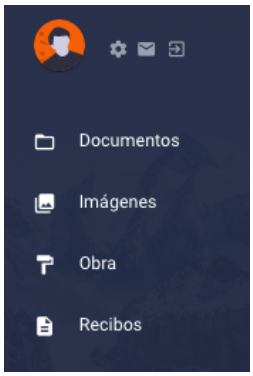
```
ngOnInit() {
   this.iconTypeMenuTitle = this.navService.iconTypeMenuTitle;
   this.menuItemsSub = this.navService.menuItems$.subscribe((menuItem) => {
     let role = localStorage.getItem("role");
     const role_clr = role.replace(/"/g, "");

     let search = "";
     if (role_clr === "admin") search = "Administrador";
     if (role_clr === "cliente") search = "Cliente";
     if (role_clr === "trabajador") search = "Trabajador";

     this.menuItems = menuItem.filter((item) => item.tooltip === search);

     //Checks item list has any icon type.
     this.hasIconTypeMenuItem = !!this.menuItems.filter(
       (item) => item.type === "icon"
     ).length;
   });

   this.layoutConf = this.layout.layoutConf;
}
```

En el método *ngOnInit* del componente *sidebar.component.ts* cargamos las rutas de los usuarios identificados en el sistema, como se puede observar, después de obtener los enlaces, mediante el *token* filtramos estos, de esta manera, solo nos quedamos con aquellas rutas que pertenecen al rol en específico.

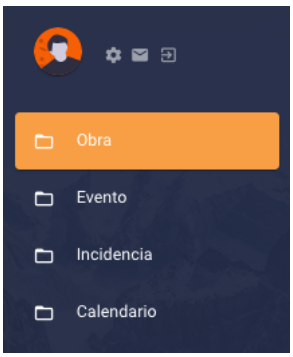
Rol Administrador



Rol Cliente



Rol Trabajador



Nota: Si un usuario intenta acceder a una ruta que no le pertenece, por ejemplo, si un cliente intenta acceder a alguna ruta del administrador, este será devuelto a la página de inicio de sesión y su *token* será borrado, lo que le obligará volver a identificarse.

4.4.4.3. *Guards*

Para poder proteger el acceso hacia las rutas creamos los *Guards*, de estos tenemos dos tipos:

- **AuthGuard:** Este tipo de *guard* se encarga de comprobar si el usuario está autenticado mediante el *token* almacenado en el *localStorage*.

```
@Injectable()
export class AuthGuard implements CanActivate{

  constructor(private router: Router, private jwtAuth: JwtAuthService) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    if (this.jwtAuth.isLoggedIn()) {
      return true;
    } else {
      this.router.navigate(["/sessions/signin"], {
        queryParams: {
          return: state.url
        }
      });
      return false;
    }
  }
}
```

Como podemos ver, el *AuthGuard* hereda una clase llamada *CanActivate* que viene proporcionada por el paquete *@angular/rou*ter.
En el constructor inyectamos los servicios necesarios para tener el control de las rutas.
En el método *canActivate* devolvemos el resultado del método *isLoggerIn* del servicio *jwtAuth*.

```
isLoggedIn(): Boolean {
  const token = this.getJwtToken();
  if(this.jwtHelper.isTokenExpired(token) || !token){
    return false;
  }
  return true;
}

getJwtToken() {
  return this.ls.getItem('token');
}
```

Los métodos invocados por el *AuthGuard* se encuentran en el servicio *jwt-auth.service.ts*.
La función *isLoggedIn* devuelve un resultado de tipo *boolean* que depende del resultado de *jwtHelper.isTokenExpired* que, al mismo tiempo, verifica el estado del *token* obtenido del usuario. Esta función pertenece al paquete *@auth0/angular-jwt*.

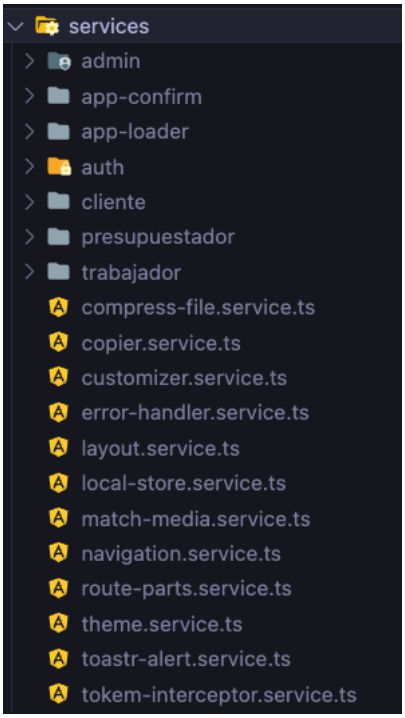
- **RoleGuard:** El *guard* de rol, además de comprobar la autenticación del usuario, también comprueba el rol que tiene asignado. Para ilustrar este funcionamiento tomamos como ejemplo el *AdminGuard*, como su nombre nos indica, comprueba si el usuario que intenta acceder a una ruta es de tipo Administrador.

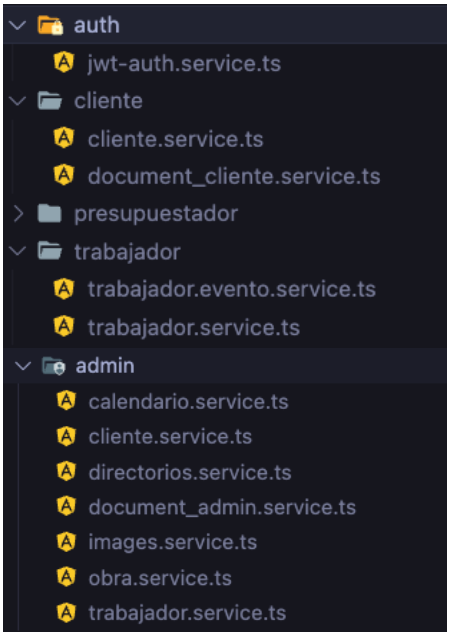
```
@Injectable({
  providedIn: 'root'
})
export class AdminGuard implements CanActivate {

  constructor(
    private jwtAuth: JwtAuthService,
    public router: Router,
    private ls: LocalStoreService,
  ){}

  canActivate(rouet: ActivatedRouteSnapshot): boolean{
    //const role = rouet.data.admin;
    const token = this.ls.getItem('token');
    const value_toke:any = decode(token)
    const role_token = value_toke.role;

    if( !this.jwtAuth.isLoggedIn() || "admin" =/= role_token ){
      this.router.navigateByUrl("/sessions/signin");
      return false;
    }
    return true;
  }
}

```

Como se puede observar, el *guard* de rol tiene la misma apariencia que el anterior, debido a que solamente cambia la lógica. En este caso, posteriormente a la recuperación del *token*, se decodifica para obtener el tipo de rol guardado por el *token*. Una vez obtenido, este es comparado con el rol exigido por la ruta.

4.4.4.4. *Servicios*

Una funcionalidad muy importante que debemos ver son los servicios, como habíamos mencionado en un apartado anterior, los servicios son utilizados o bien para comunicar entre componentes o bien para realizar peticiones *HTTP* entre el fornt-end y el back-end.





Hemos repartido la funcionalidad de los usuarios entre los diferentes roles, donde cada directorio hace referencia a uno en concreto. De esta forma tenemos las funciones organizadas. Los servicios generales se encuentran directamente dentro del directorio */services.*

A la hora de manejar los datos, es buena práctica utilizar el tipado que nos ofrece *TypeScript*, de este modo tenemos controlado los atributos de cada objeto en todo momento, por ejemplo:

```
1  return this.http.get<Cliente>(
```

```
1  return this.http.get<Trabajador>
```

```
1  return this.http.get<EventoCalendario[]>
```

```
1  return this.http.get<Document>
```

4.4.4.5. *Modelos*

Una de las ventajas que nos ofrece *TypeScript* es el tipado de los datos, para aprovechar este al máximo hemos ido definiendo los modelos (interfaces) de cada objeto. De este modo, al obtener un dato desde el back-end, indicamos su tipo:

4.4.4.6. *Pipes*

A la hora de mostrar algún dato en la interfaz del usuario nos puede interesar realizar un cambio tanto por estética o bien por funcionalidad. En este apartado vemos algún ejemplo de los Pipes implementados en esta aplicación.

1. En primer paso vemos la comanda empleada para crear un nuevo Pipe:
   Esta comanda nos crear un Pipe nuevo llamado *format-data* en el directorio */shared/pipes/.*

```
ng generate pipe /shared/pipes/format-data
```

2. El segundo paso es importar el Pipe en el módulo compartido (*SharedPipesModule*), de esta manera, si un componente necesita utilizarlo, este importará este módulo completo.

```
const pipes = [
  RelativeTimePipe,
  ExcerptPipe,
  GetValueByKeyPipe,
  FormatDataPipe,
];

@NgModule({
  imports: [CommonModule],
  declarations: pipes,
  exports: pipes,
})
export class SharedPipesModule {}
```

3. El tercer paso es escribir la lógica que ejecutará este Pipe, es decir, la funcionalidad que debe realizar, este este caso, formatear la fecha.

```
@Pipe({
  name: "formatData",
})
export class FormatDataPipe implements PipeTransform {
  transform(value: string) {
    var datePipe = new DatePipe("en-US");
    value = datePipe.transform(value, "dd/MM/yyyy");
    return value;
  }
}
```

En esta captura podemos observar cómo es un Pipe en Angular. La lógica del Pipe está en la función *transform*, la cual recibe un parámetro de tipo *string*, a partir de este valor realizamos unos cambios y devolvemos el nuevo valor.

4. Una vez tenemos la lógica de Pipe creada, ya podemos utilizarlo para modificar nuestros datos.

```
<mat-grid-tile [colspan]="2" [rowspan]="1" >
    <mat-form-field class="ancho-row">
        <mat-label>Nacimiento</mat-label>
        <input matInput autocomplete="off" type="text" name="birthDay | FormatDataPipe" formControlName="birthDay">
    </mat-form-field>
</mat-grid-tile>
```

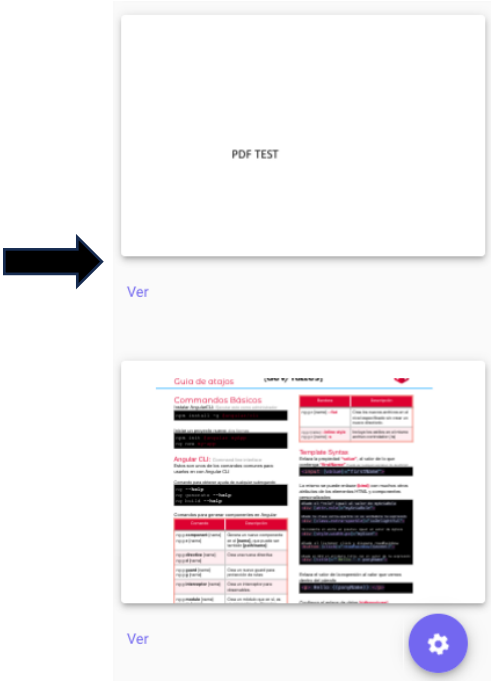
Como podemos ver en esta captura, nos situamos en el campo de fecha e invocamos al Pipe mediante "|" seguido de su nombre. De esta forma, el Pipe ya recibe automáticamente el valor a cambiar.

4.4.4.7. *Pop up*

A la hora de gestionar los diferentes componentes, como primera opción, podemos ir creando rutas y subrutas en el *sidebar*, sin embargo, esta solución a la larga puede no cumplir con el objetivo de memorabilidad, ya que será difícil para un usuario recordar todos los pasos para llegar a una funcionalidad en específico. Como alternativa, utilizamos los Pop up, mediante los cuales cargamos el componente que se desea mostrar, de esta forma es más fácil de acceder.

Para ilustrar un ejemplo tenemos varios casos, por ejemplo, permitimos a los clientes visualizar sus documentos mediante este.

```html
<div *ngIf="documentos.length != 0">
  <div fxLayout="row wrap" fxLayoutGap="16px grid">
    <div
      fxFlex="25%"
      fxFlex.xs="100%"
      fxFlex.sm="33%"
      *ngFor="let elem of documentos"
    >
      <mat-card class="mat-elevation-z4">
        <div class="pdf-card">
          <pdf-viewer
            [src]="elem.url"
            [page]="1"
            [render-text]="true"
            [original-size]="false"
            style="width: 300px; height: 300px"
          ></pdf-viewer>
        </div>
      </mat-card>
      <mat-card-actions style="display: flex">
        <button mat-button color="warn" (click)="deleteDocument(elem)">
          <mat-icon>delete</mat-icon>
        </button>
        <button mat-button color="primary" (click)="verPDF(elem)">Ver</button>
      </mat-card-actions>
    </div>
  </div>
</div>
```



Mediante el código mostrado, logramos enseñar los documentos que el cliente tiene en una obra específica. Estos documentos se muestran en forma de cartas (*Cards*), de esta manera, conseguimos un diseño limpio. También permitimos al usuario visualizar el contenido del documento mediante la miniatura.

Cada *item* (documento), tiene asignado un botón *Ver* para visualizar dicho documento de forma completa. Este botón al mismo tiempo está asociado a un método llamado **verPDF** *(de tipo onClickListener*) que recibe como argumento el documento seleccionado.

```typescript
verPDF(item) {
  const dialogRef = this.dialog.open(PdfViewerComponent, {
    width: "80%",
    height: "80%",
    data: {
      document: item.url,
    },
  });

  dialogRef.afterClosed().subscribe((res) => {});
}
```

Al hacer clic en el botón *Ver* de un documento, el método *verPDF* se encarga de crear un *dialogo* (Pop up) sobre el cual carga el componente *PdfViewerComponente* con una serie de estilos y, sobre todo, le pasamos parámetros (la URL a mostrar).

Para poder utilizar los diálogos, importamos el módulo *DialogModule* de la librería *primeng/dialog*.

```
@Component({
  selector: "app-pdf-viewer",
  template: `
    <pdf-viewer
      [src]="pdfSrc"
      [page]="1"
      [render-text]="true"
      [original-size]="false"
      style="width: 100%; height: 100%;"
    ></pdf-viewer>
  `,
})
export class PdfViewerComponent implements OnInit {
  pdfSrc: string;
  constructor(
    private router: Router,
    private rout: ActivatedRoute,
    @Inject(MAT_DIALOG_DATA) public data
  ) {
    this.pdfSrc = data.document;
  }

  ngOnInit(): void {}
}
```

En la captura presentada, podemos ver el contenido del componente *PdfViewerComponent*. Para poder recuperar el valor de la URL, debemos inyectar un servicio específico en el constructor de este, como consecuencia de este hecho, ya podemos extraer el campo *document* del objeto *data*.
Para poder visualizar los documentos en forma de PDF en nuestra aplicación, utilizamos la librería *ng2-pdf-viewer*.

4.4.4.8. *Vistas aplicación administrativa*

En este apartado vemos algunas vistas de ejemplo de la aplicación administrativa:

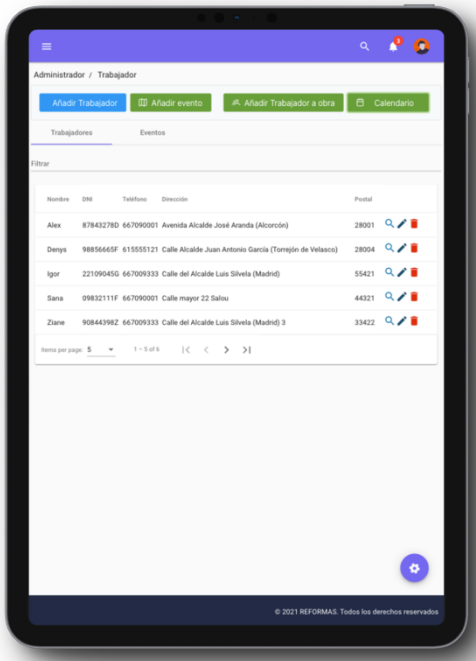


***Figura 36**. Vista trabajadores - Rol Administrador.*

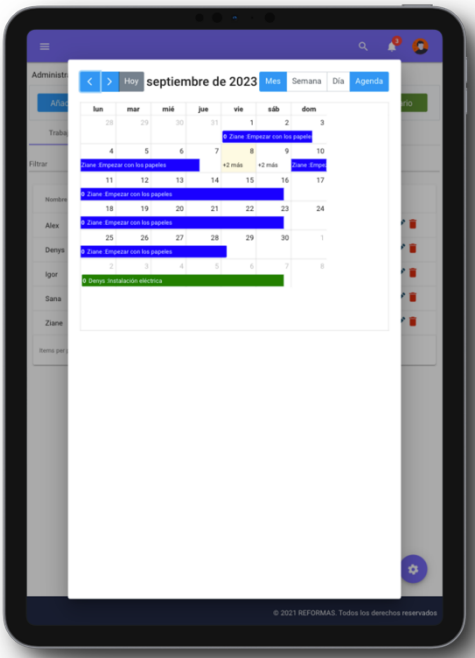


Figura 37.Vista calendario - Rol Administrador.

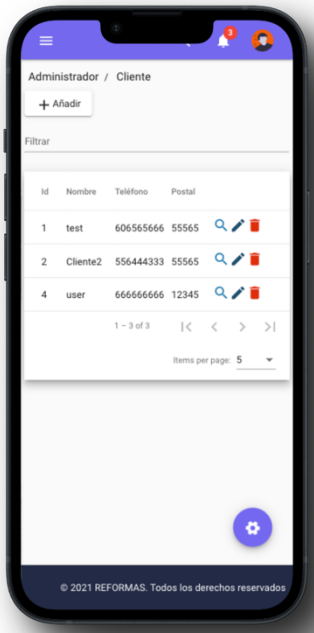
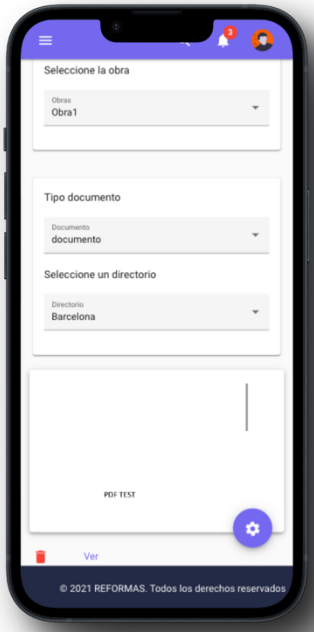


***Figura 38**. Vista cliente - Rol Administrador.*



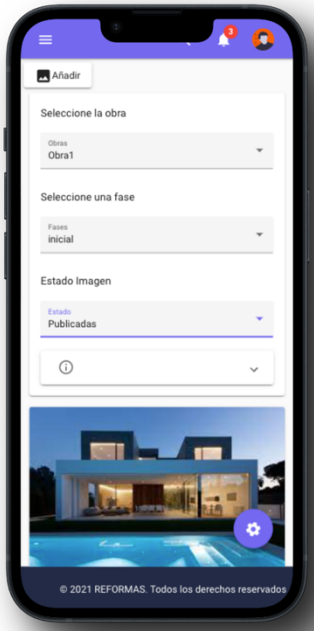
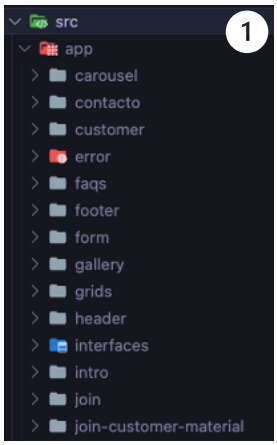


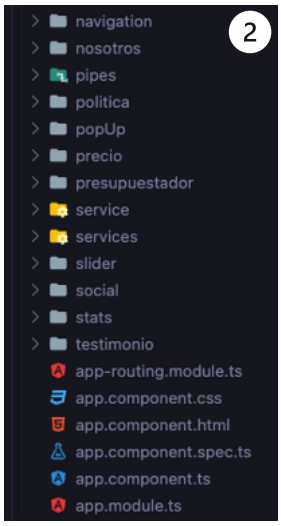
***Figura 40**. Vista imágenes - Rol Administrador.*

***Figura 41**. Vista obra - Rol Administrador.*

4.5. Implementación front-end sistema presupuestación

Para la creación de la aplicación del sistema de presupuestación, también hemos utilizado el framework de Angular. Para evitar la repetición de conceptos, solamente comentamos aquellos aspectos que consideramos importantes.
Nota: La complejidad a la que nos hemos enfrontado a la hora de implementar este sistema de presupuestación es como interpretar y transformar una decena de hojas de cálculo (Excel) en una aplicación Angular intuitiva y fácil de utilizar.





A parte del sistema de presupuestación, este proyecto contiene una *landing page*[3] donde la empresa pone a disposición del cliente información sobre los servicios que ofrece.
En las capturas mostradas, podemos ver aquellos componentes que forman parte de esta. El componente para destacar es el llamado *presupuestador*, ya que contiene aquellas clases, modelos relacionados con el sistema de presupuestación.

4.5.1. Sistema de presupuestación

El sistema de presupuestación contiene un conjunto de clases que, dado unos precios y medidas, estas se encargan de calcular el coste de una reforma. A continuación, entramos más en detalle.

Como podemos observar en esta captura, el directorio *presupuestador* contiene:

- **Class**: En este directorio, tenemos creado todas las clases que usamos para obtener el coste final de la reforma, donde cada una de estas clases se encarga de calcular solamente el precio de una parte.
- **Interfaces**: En la carpeta interfaces, es donde declaramos los modelos que usamos para saber qué atributos tiene cada objeto.
- **Model**: Contiene objetos instanciables necesarios para implementar este sistema.
- **Reforma-integral**: Es el componente que se encarga de invocar aquellas clases correspondientes y sumar el coste final que estas devuelven.
- **Otros**: Los ficheros que restan pertenecen al componente *presupuestador.component*, su utilizar es permitir al cliente seleccionar las partes que desea reformar.

[3] https://blog.hubspot.es/website/landing-page

4.5.2. Componente inicio Landing Page.

Figura 42. *Página inicio del landing page.*

En el componente *home*, el usuario o bien accede a la aplicación administrativa, o bien entra en el presupuestador. Al acceder al sistema de presupuestación, el primero componente ofrece al usuario mediante un menú, escoger aquellas zonas de su casa que desea restaurar. Gracias a la división de la reforma en parte, el cliente solamente tendrá que introducir datos de aquellas partes seleccionadas.

4.5.3. Componente inicio Presupuestador.

Las siguientes figuras a mostrar pertenecen al componente principal del presupuestador, están tomadas utilizando un navegador Google Chrome versión 116.0.5845.140 con una arquitectura arm64.

***Figura 43**. Vista ordenador MacBook-Pro.*

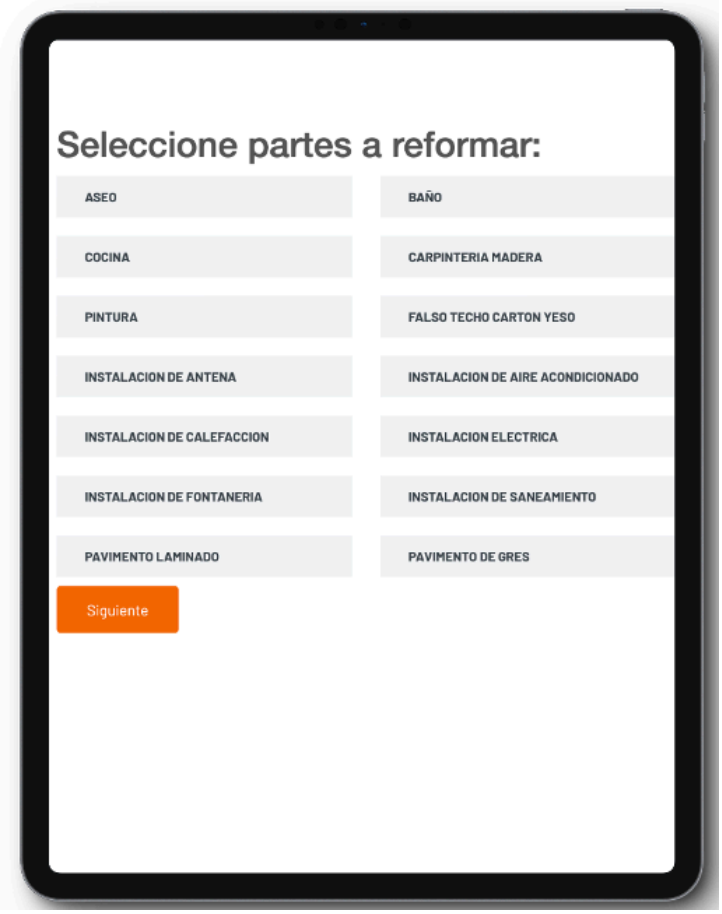


Figura 45. Vista iPad 11.

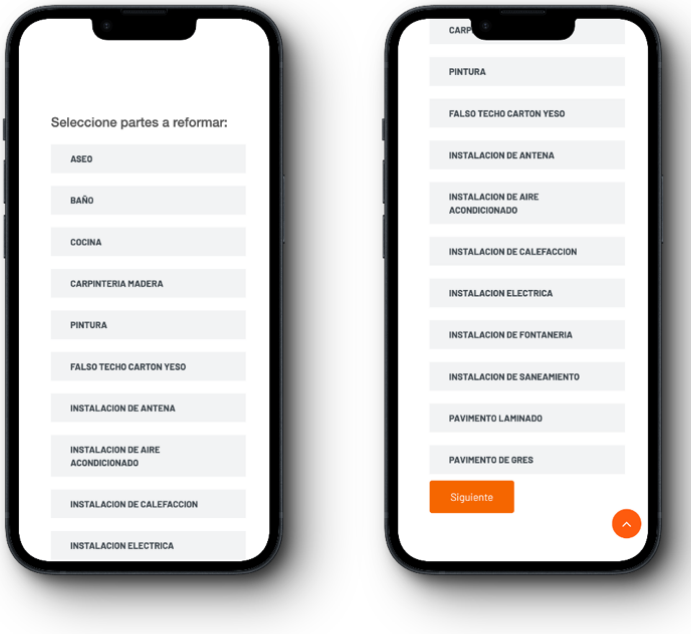


Figura 44. *Vista iPhone 14.*

4.5.4. Componente reforma-integral

Una vez seleccionadas las partes que se desea reformar, se procederá a introducir algún dato obligatorio. Solamente se solicitará los valores de aquellos componentes escogidos, de esta forma, no agobiaremos al cliente con información innecesaria.

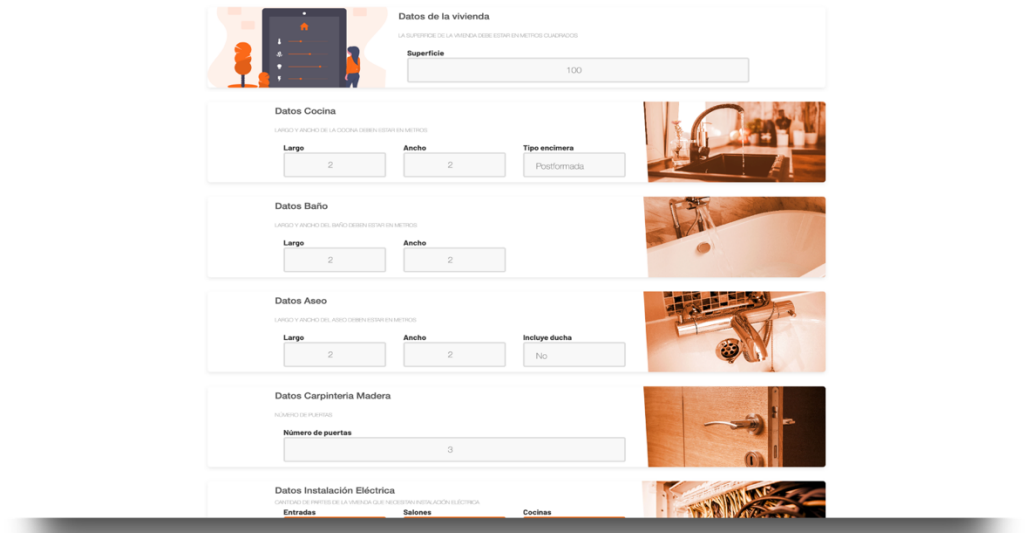


***Figura 46**. Vista ordenador MacBook-Pro del Componente reforma-integral 1.*

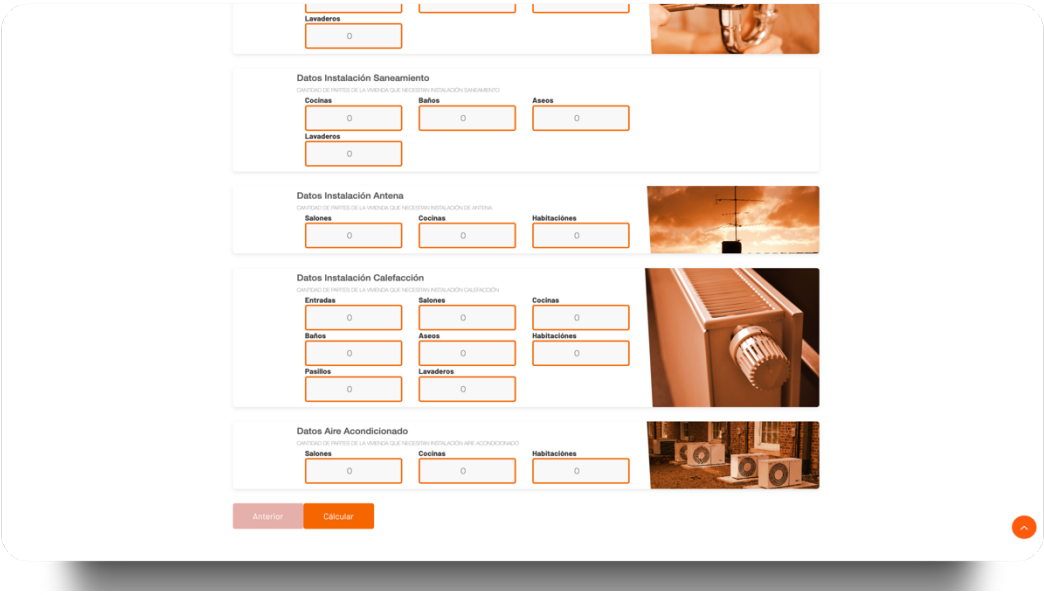


***Figura 47**. Vista ordenador MacBook-Pro del Componente reforma-integral 2.*

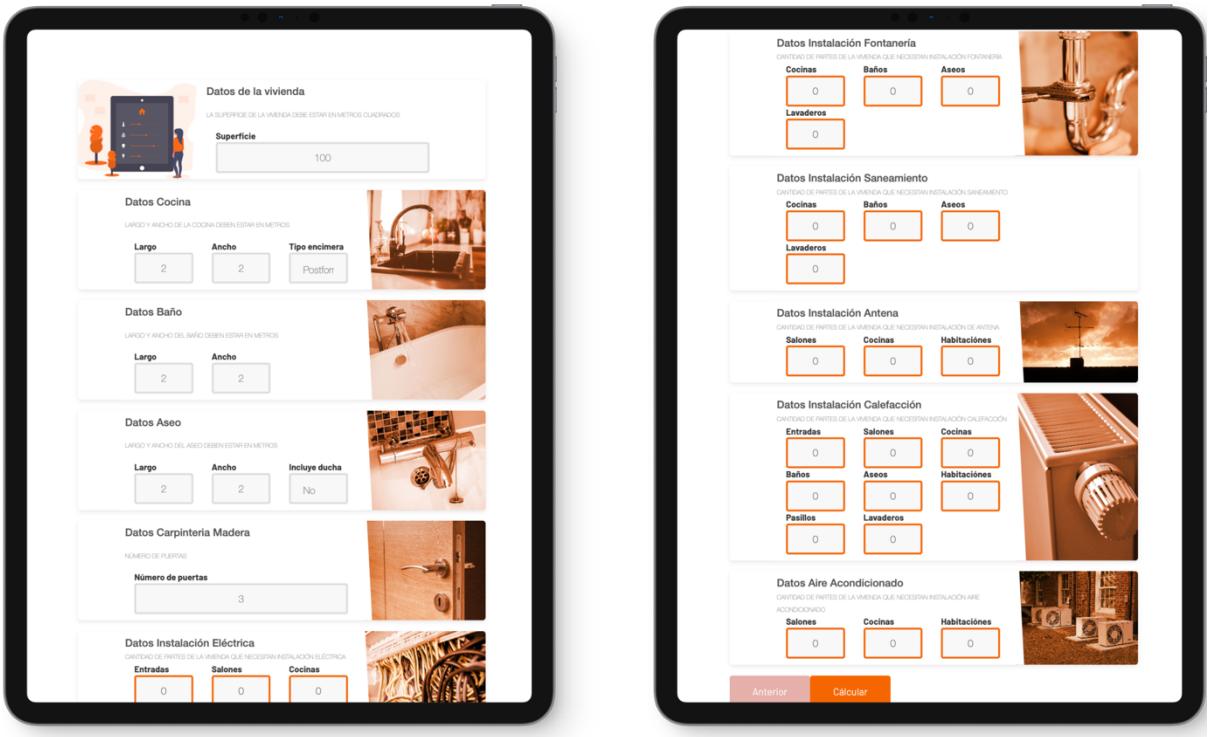


Figura 48. *Vista iPad 11 del Componente reforma-integral.*

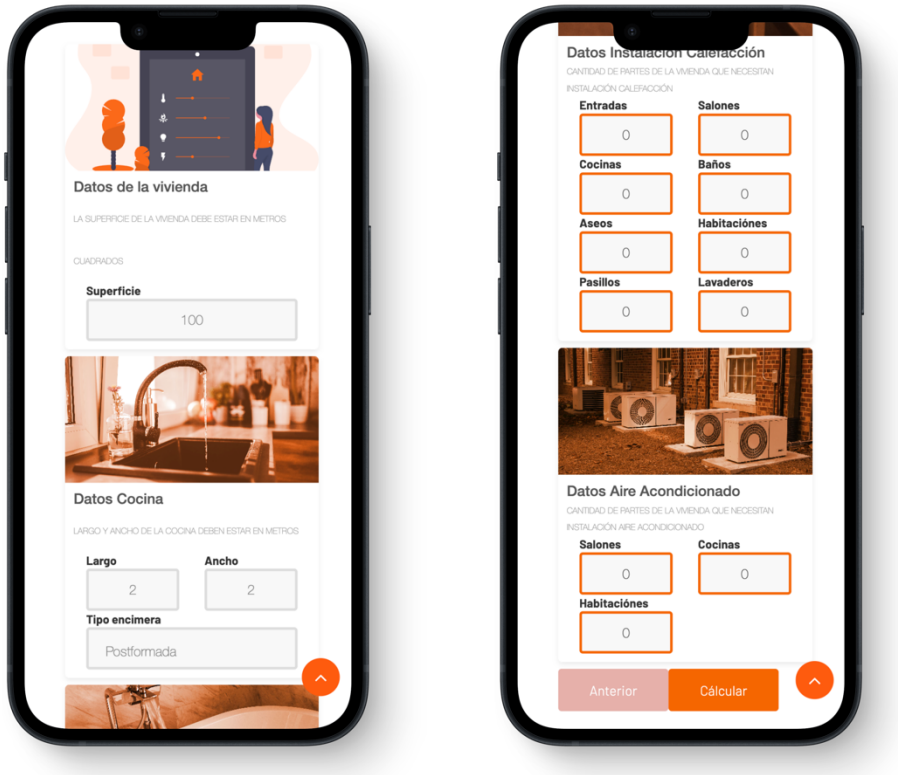


Figura 49. *Vista iPhone 14 del Componente reforma-integral.*

Como podemos notar, el botón *Calcular* estará oculto hasta que se rellenen los campos obligatorios. También el formulario incluye la validación de los campos, es decir, no se podrá introducir caracteres especiales (para evitar inyección *SQL*), tampoco se aceptan valores negativos, en número de caracteres de cada valor está limitado, entre otras características.

Después de introducir las medidas y el formulario es enviado, nuestra aplicación tardará menos de 10 segundos para realizar los cálculos y nos mostrará una nueva vista con el precio total que costará la reforma, asimismo, nos enseñará el precio de cada zona.

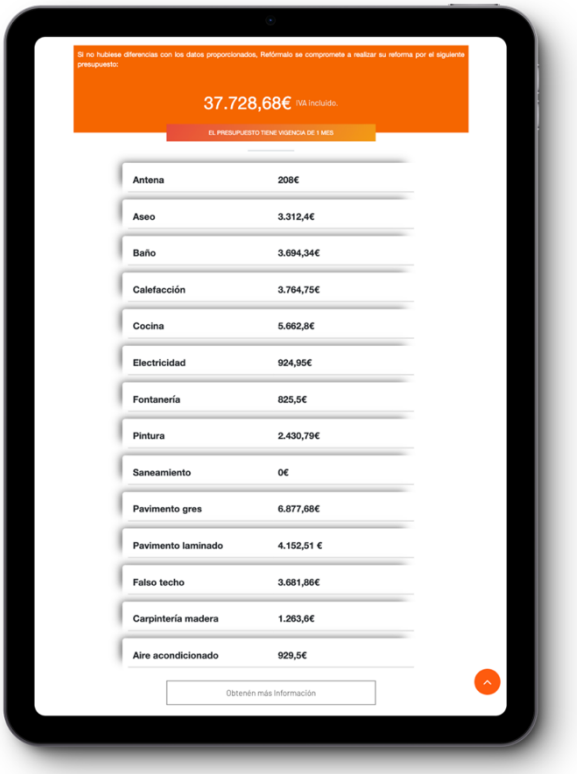


Figura 50. *Vista iPad 11 del Componente reforma-integral - precio.*

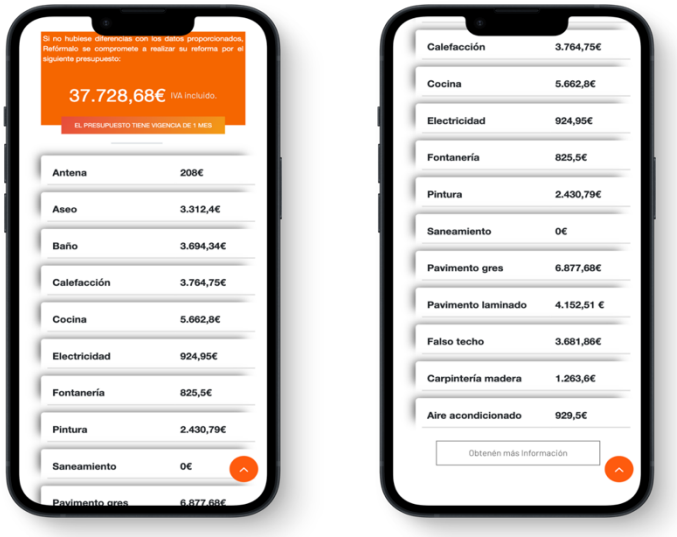


Figura 51. Vista iPhone 14 del Componente reforma-integral - precio.

Como podemos notar en las anteriores figuras, disponemos de un botón *obtén más información* que permite al usuario obtener un documento PDF más detallado. Este documento será enviado a la cuenta de correo electrónico introducido en el formulario que se mostrará al hacer clic encime de este botón.

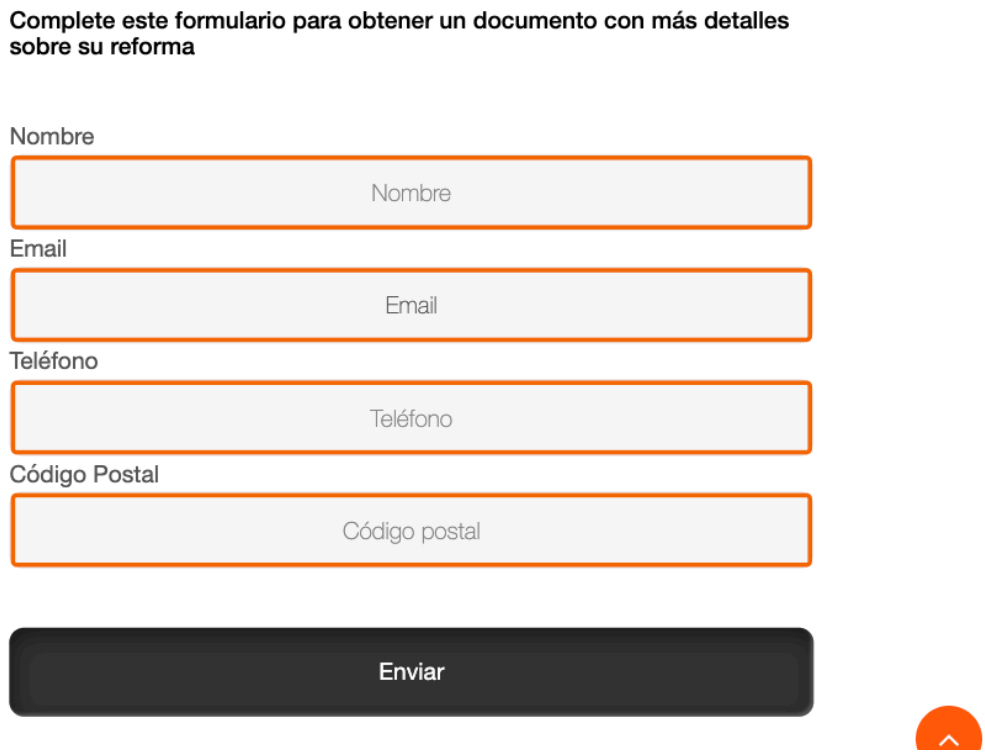


***Figura 52**. Vista del formulario para solicitar el documento PDF.*

El botón *Enviar* es el encargado de realizar la petición al back-end comentado anteriormente, para generar el documento PDF pasándole las medidas introducidas y los precios ofrecidos. Finalmente, este documento será adjuntado al correo facilitado. Cabe recordar que estos valores serán guardados en nuestras bases de datos y la empresa se pondrá en contacto con el cliente para proceder con la reforma.

5. Evaluación

En esta sección de evaluación procedemos a evaluar los diferentes aspectos del sistema implementado, tanto los requisitos funcionales como no funcionales.

Para esta evaluación hemos utilizado un ordenador MacBook Pro con capacidad 8GB de RAM y con un sistema operativo Ventura 13.5. Un navegador Google Chrome versión 116.0.5845.140 con una arquitectura arm64.

Para mostrar toda la funcionalidad conseguida, como hemos visto en anteriores apartados, dividiremos la aplicación administrativa en los diferentes roles (administrados, cliente y trabajador).

***Tabla 6**. Evaluación tareas del sistema presupuestación. Elaboración propia.*

Índice	Tarea	Ordenador	Tablet	Móvil	OK	N.º Clics
1	Seleccionar las partes a reformar.	✓	✓	✓	✓	1
2	Introducir las medidas de aquellas partes seleccionadas.	✓	✓	✓	✓	2
3	Generar el coste de cada parte.	✓	✓	✓	✓	3
4	Mostrar el coste total de la reforma.	✓	✓	✓	✓	3
5	Permitir al usuario recibir un documento PDF con los detalles.	✓	✓	✓	✓	4
6	Generar el presupuesto en forma de PDF.	✓	✓	✓	✓	-
7	Enviar el presupuesto al correo del usuario.	✓	✓	✓	✓	-
8	Guardar un registro del presupuesto dado.	✓	✓	✓	✓	-

Todas las funcionas que el cliente ha marcado se han implementado ajustando a sus deseos y cumpliendo al máximo sus expectativas, gracias a las diferentes reuniones que hemos ido teniendo durante y después del ciclo del desarrollo.
El número de clics viene a mostrar los pasos que realiza el usuario para llegar a una acción. Gracias al diseño de la aplicación, el manejo de esta es intuitivo, y el usuario solo tiene que darle a siguiente. Los pasos 6-8 se realizan automáticamente después del paso 4.

***Tabla 7**. Evaluación tareas de la aplicación administrativa rol Administrador. Elaboración Propia.*

Índice	Tarea	Ordenador	Tablet	Móvil	OK	N.º Clics
1	Log in.	✓	✓	✓	✓	1
2	Alta cliente.	✓	✓	✓	✓	2
3	Alta trabajador.	✓	✓	✓	✓	2
4	Alta obra.	✓	✓	✓	✓	2
5	Subir foto.	✓	✓	✓	✓	2
6	Publicar foto.	✓	✓	✓	✓	3
7	Crear evento.	✓	✓	✓	✓	2
8	Asignar evento al trabajador	✓	✓	✓	✓	2

***Tabla 8**. Evaluación tareas de la aplicación administrativa rol Trabajador. Elaboración Propia.*

Índice	Tarea	Ordenador	Tablet	Móvil	OK	N.º Clics
1	Log in.	✓	✓	✓	✓	1
2	Listar eventos.	✓	✓	✓	✓	2
3	Notificar incidencia.	✓	✓	✓	✓	2

***Tabla 9**. Evaluación tareas de la aplicación administrativa rol Cliente. Elaboración Propia.*

Índice	Tarea	Ordenador	Tablet	Móvil	OK	N.º Clics
1	Log in.	✓	✓	✓	✓	1
2	Ver imágenes obra	✓	✓	✓	✓	2
3	Ver documentos.	✓	✓	✓	✓	2
4	Ver planos.	✓	✓	✓	✓	2

***Tabla 10**. Evaluación tareas de la aplicación administrativa rol Trabajador. Elaboración Propia.*

Índice	Tarea	Descripción
1	Vistas intuitivas.	Para conseguir vistas intuitivas, hemos utilizado componentes de otras bibliotecas como *Angular material* o *PrimeNG*. En relación con los componentes propios, gracias a las propiedades de CSS como *Flexbox* y las @media, conseguimos que estos sean *responsive* y con un estilo apropiado.
2	Responsive.	Las vistas se tienen que adaptar a diferentes pantallas de diferentes tamaños.
3	Navegación intuitiva.	Para realizar una acción, esta tiene que resultarle fácil de encontrar al usuario, es decir, el usuario debe llegar a dicha acción en pocos pasos.
4	Capacidad de carga y eficiencia.	Gracias al *lazy load* de Angular, conseguimos optimizar la carga de nuestra aplicación. Cabe destacar que todos los ficheros vienen servidos por el servidor Cloud de S3, lo que ayuda con la carga de estos.
5	Autenticación.	Para proteger el acceso a los diferentes recursos, hemos utilizado *tokens*, los cuales se obtienen al autenticarse con el servidor back-end. Para verificar este acceso al intentar acceder a una ruta, usamos los *Guards*.
7	Cumplir con GDPR	Para cumplir con la normativa general de protección de datos europea, hemos subido nuestros datos a la plataforma de *AWS* seleccionado zonas dentro de Europa.

Por último, hemos utilizado la herramienta *Lighthouse* para analizar el rendimiento de esta, el cual nos ha dado buenos resultado.

6. Despliegue

En esta sección mostramos los pasos que se han llevado a cabo para desplegar las diferentes aplicaciones en producción. Como plataforma de despegue utilizamos los servicios ofrecidos por *AWS*.

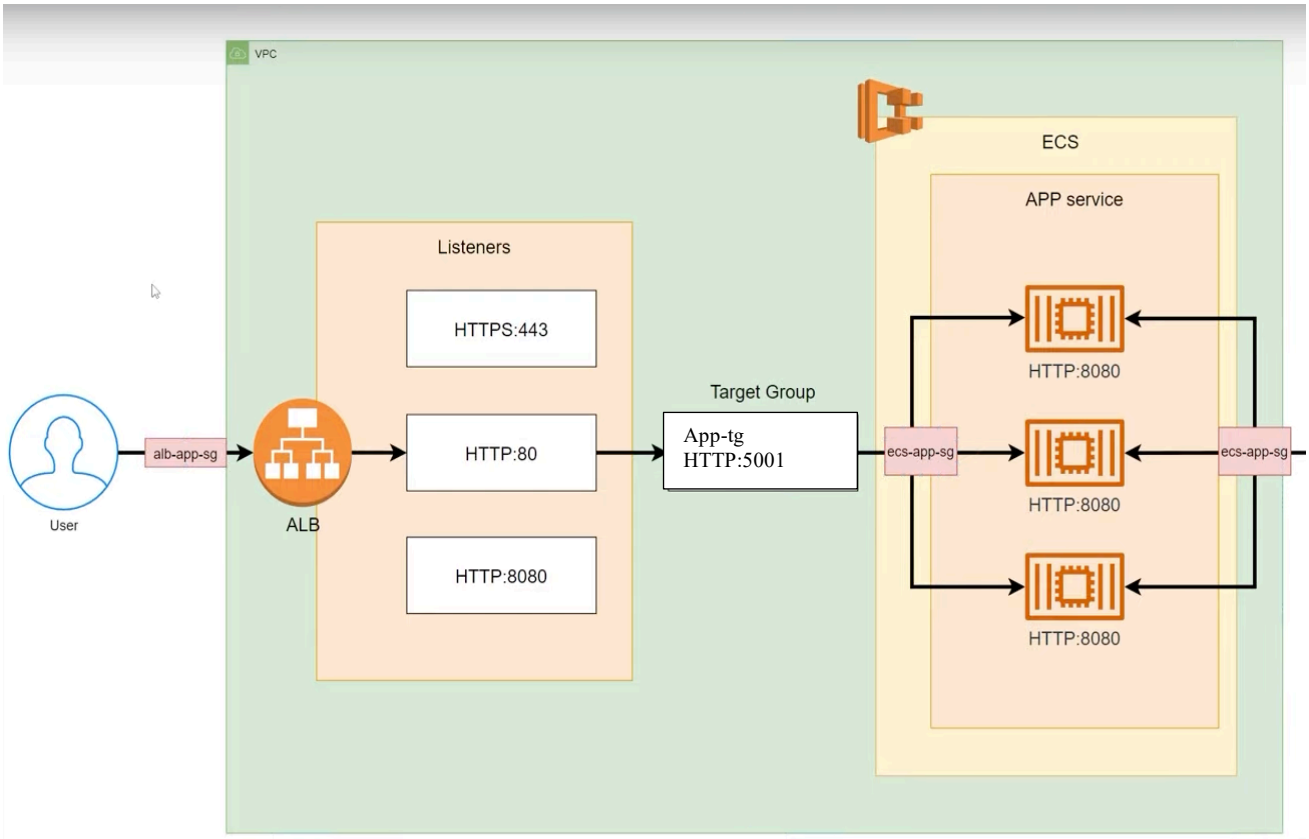


Figura 53. *Figura AWS Fargate.*

https://www.youtube.com/watch?v=7qvc1WgPNH4&t=1537s

En la Figura 53 podemos observar cómo está construido el sistema de despliegue de la *API* de la aplicación administrativa.

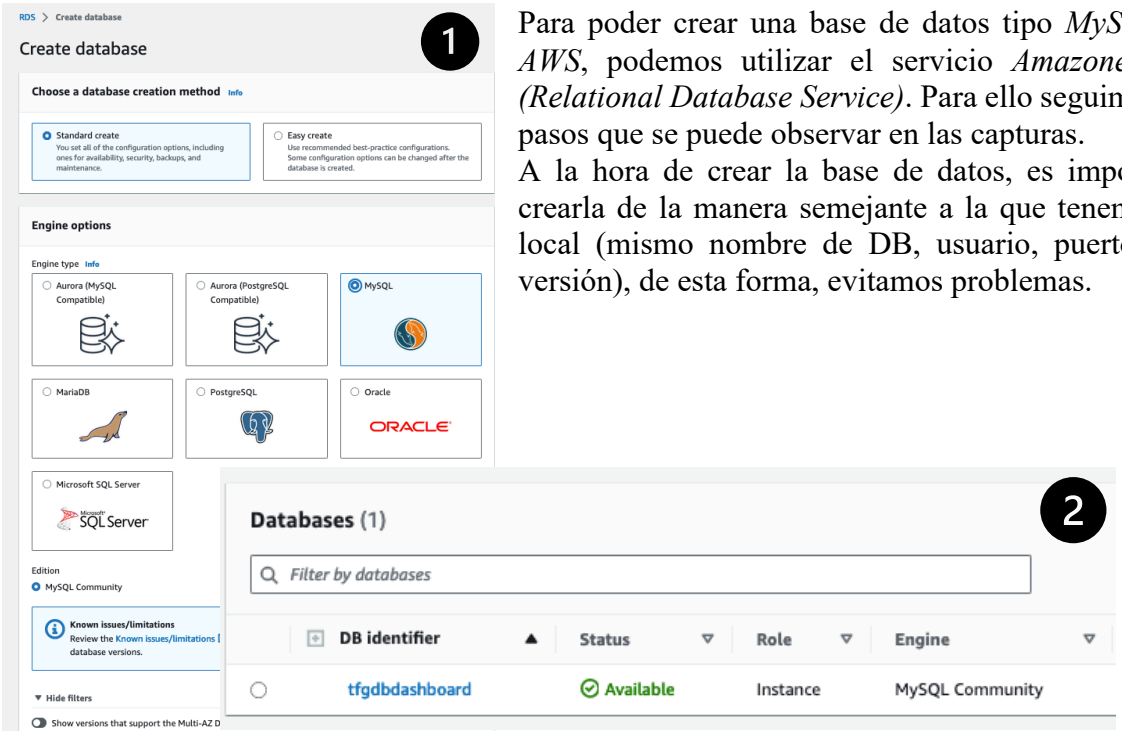
1. El usuario al solicitar una petición se encuentra con el primer *firewall* (*security group*), el cual comprueba el puerto utilizado para acceder, este tiene que ser 80.
2. Los demás elementos se encuentran dentro de una red virtual (*VPC*) creada en *AWS*.
   El *load balancer* se encarga de distribuir la carga de tráfico entre los contenedores réplica que hay disponibles. Podemos poner a disposición del cliente varios puertos de entrada, pero en este caso, utilizaremos el 80.
3. Mediante el *target group*, podemos definir los recursos que el servicio *Fargate* ofrecerá a los usuarios. *Fargate* se encargará de crear las diferentes réplicas de contenedores utilizando como imagen de *Docker* una almacenada en *ECS*.

La estructura presentada es la misma para las diferentes aplicaciones, solamente cambia la imagen de *Docker* y el puerto.

3.1. Creación de bases de datos

Antes de comenzar con el despliegue, debemos crear las bases de datos en *AWS* y conectar nuestros back-ends a estas.

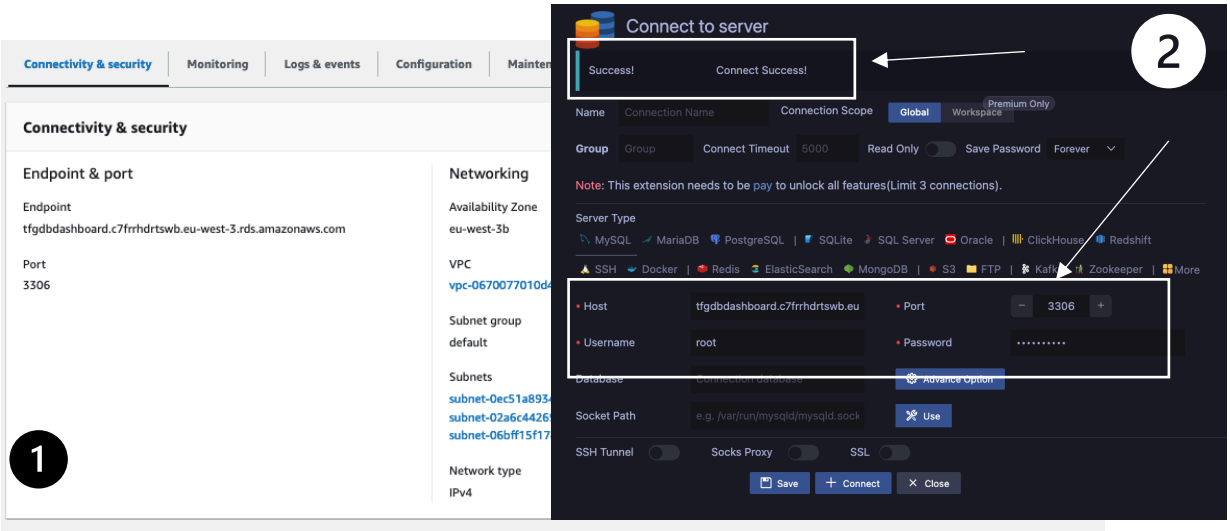
3.1.1. Base de datos del sistema administrador- Amazon RDS



Para poder crear una base de datos tipo *MySQL* en *AWS*, podemos utilizar el servicio *Amazone RDS (Relational Database Service)*. Para ello seguimos los pasos que se puede observar en las capturas.
A la hora de crear la base de datos, es importante crearla de la manera semejante a la que tenemos en local (mismo nombre de DB, usuario, puerto y la versión), de esta forma, evitamos problemas.

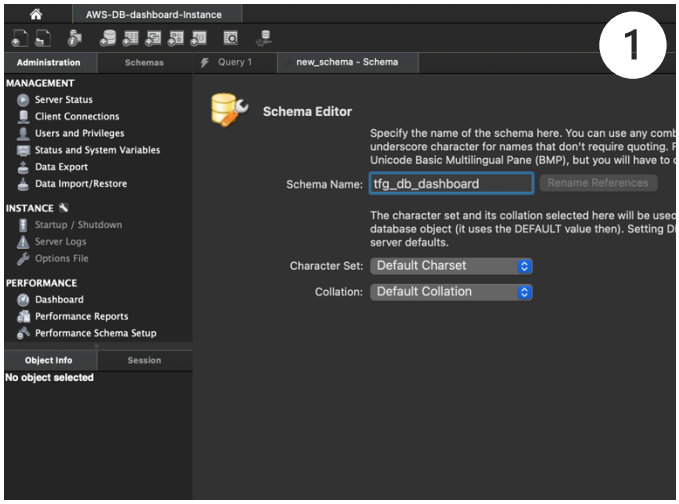
Una vez creada la instancia de base de datos, es muy importante añadir correctamente los grupos de seguridad, ya que, de lo contrario no podremos acceder a ella desde nuestros back-ends.

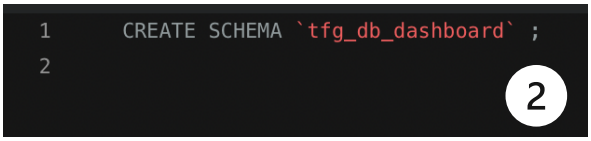
Para verificar el acceso a instancia creada, podemos acceder a esa desde nuestra máquina utilizando las credenciales ofrecidas por la instancia en la consola *AWS*:

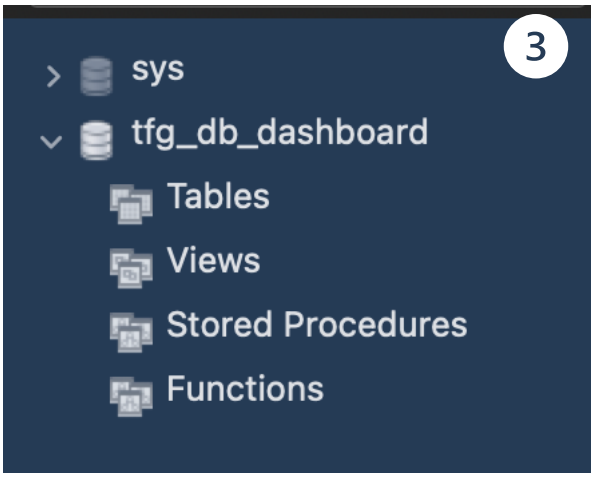


Una vez verificado el acceso ya podemos conectar la aplicación back-end a esta.

Nota: Es importante saber que en el paso anterior hemos creado una instancia, por lo tanto, después de conectarse, debemos crear la base de datos correspondiente (*Schema*):







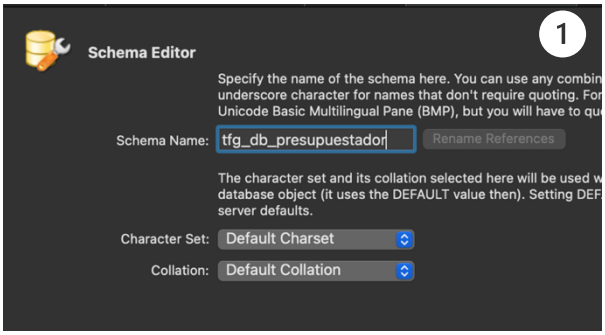
Como último paso, debemos sustituir las credencias locales por las nuevas de la instancia de la base de datos *RDS* de *AWS* (en el fichero *.env*). De esta forma ya tenemos conectada nuestra aplicación con el servicio de Amazon.

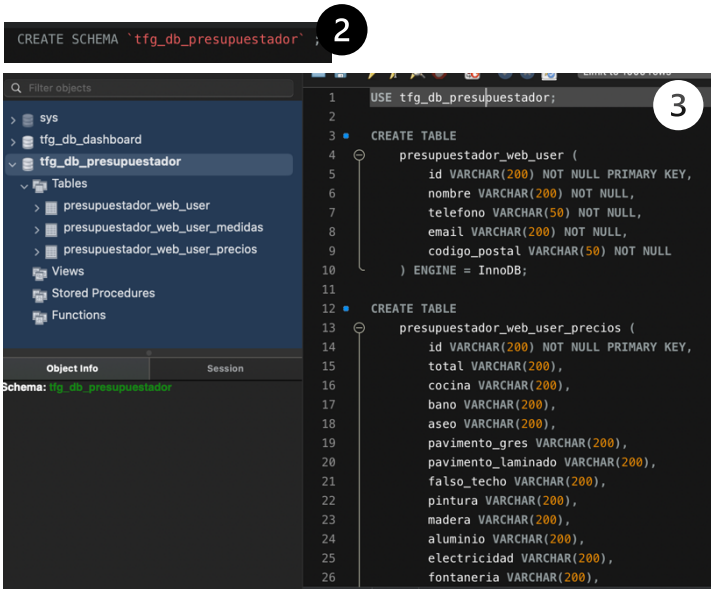
3.1.2. Base de datos sistema presupuestación

Como hemos visto en el apartado de **Implementación back-end sistema presupuestación**, el sistema de presupuestación utiliza dos tipos de bases de datos:

- Una base de datos relacional para almacenar las medidas que el usuario ha introducido, el presupuesto que nuestra web ha calculado y finalmente, los datos de contacto del cliente interesado.
- La segunda, es una base de datos *NoSQL* para almacenar los precios y porcentajes utilizados por el sistema de presupuestación para realizar los cálculos.
  Nota: Esta base de datos ya está creada y conectada en el punto comentado.

Dentro de la misma instancia creada en el punto anterior, creamos una nueva base de datos. De esta manera, tendremos una base de datos para cada aplicación, lo que nos lleva a dividir tanto la carga como datos.





Para crear las tablas, utilizamos el código *SQL* que teníamos. Una vez acabado con las bases de datos necesarias, entonces podemos continuar con el despliegue.

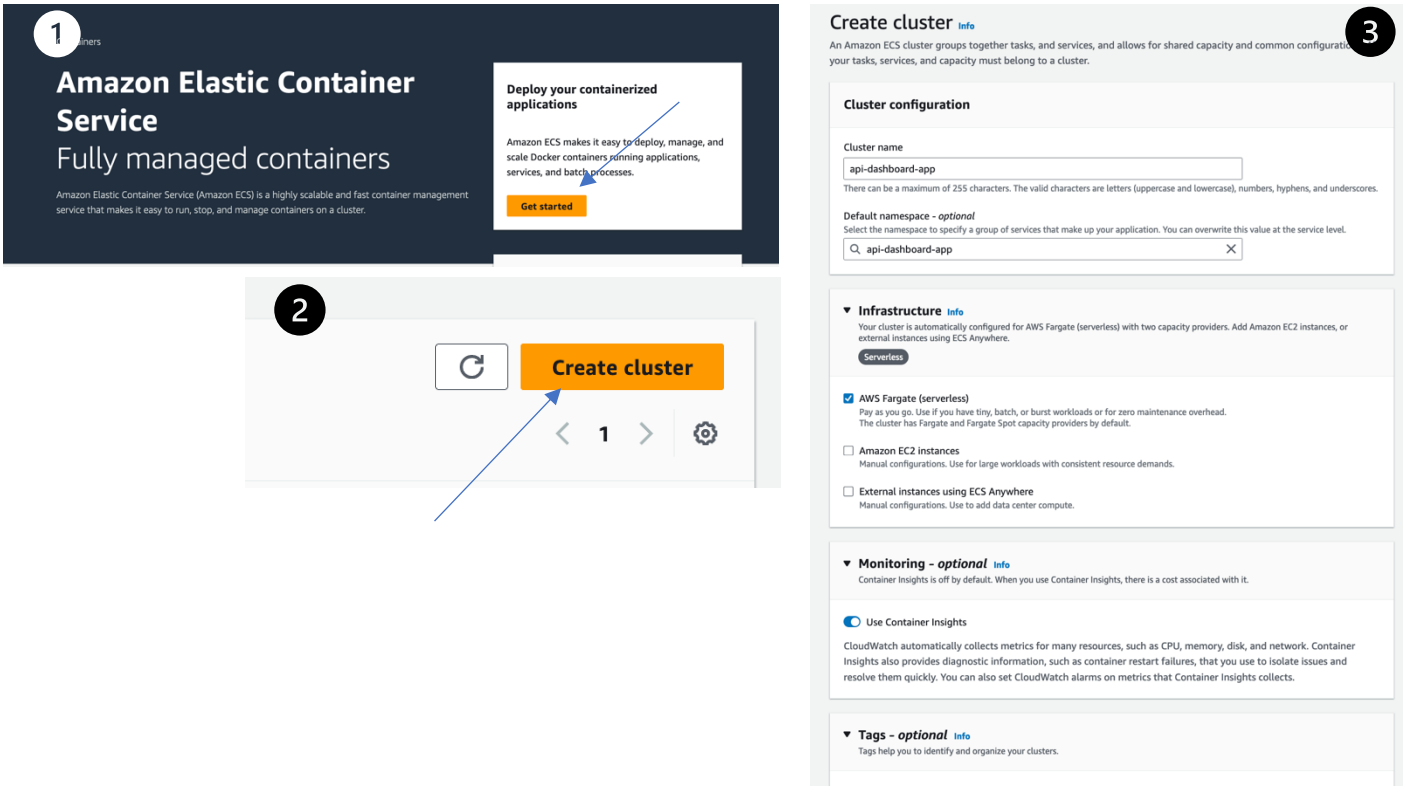
3.2. Amazon Fargate

A la hora de subir nuestras aplicaciones a la nube, disponemos de varias maneras y herramientas para hacerlo. En nuestro caso, aprovechando el hecho de tener las aplicaciones englobadas en contenedores de *Docker*, utilizamos algún servicio de *AWS* que permita trabajar con estos.

Dentro de Amazon, se destacan dos servicios:

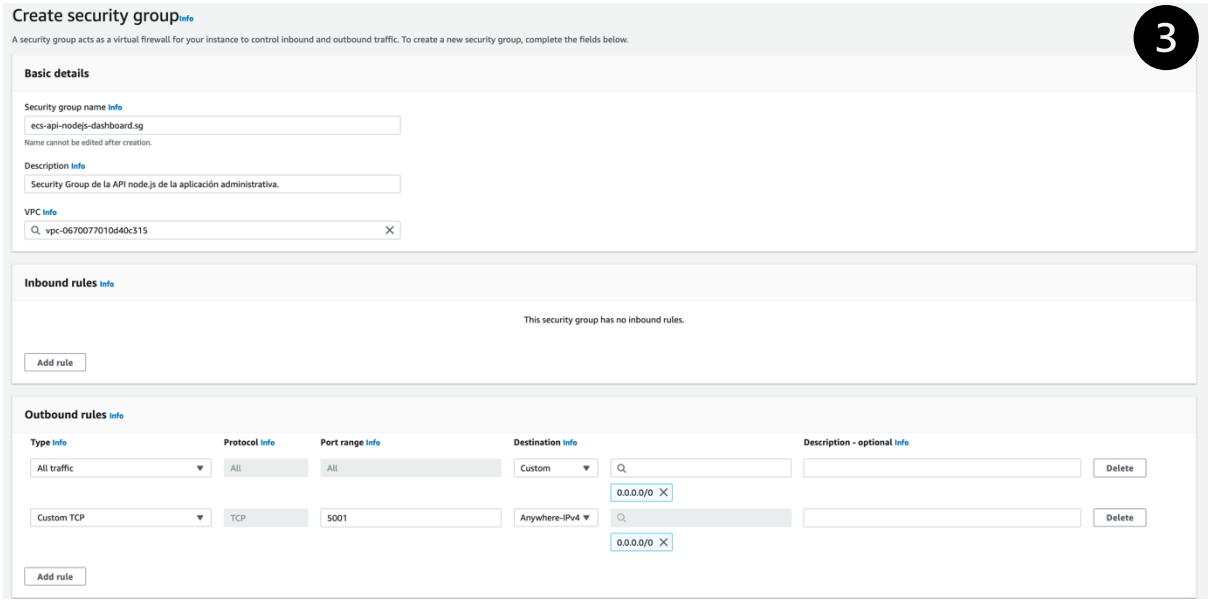
- **Amazon ECS** (Elastic Container Service) [42]: Es un servicio de orquestación de contenedores que facilita la implementación, administración y, sobre todo, el escalado de las aplicaciones. Este servicio ofrece configuración y prácticas recomendadas operativas integradas de *AWS*. También se puede usar junto a otros servicios como *Amazon ECR* o *Docker*.
- **Amazon Fargate** [43]: La tecnología *AWS Fargate* internamente utiliza el servicio *ECS* para ejecutar contenedores, pero sin tener que administrar servidores ni clústeres de las instancias de *Amazon EC2* (Máquinas virtuales de Amazon).

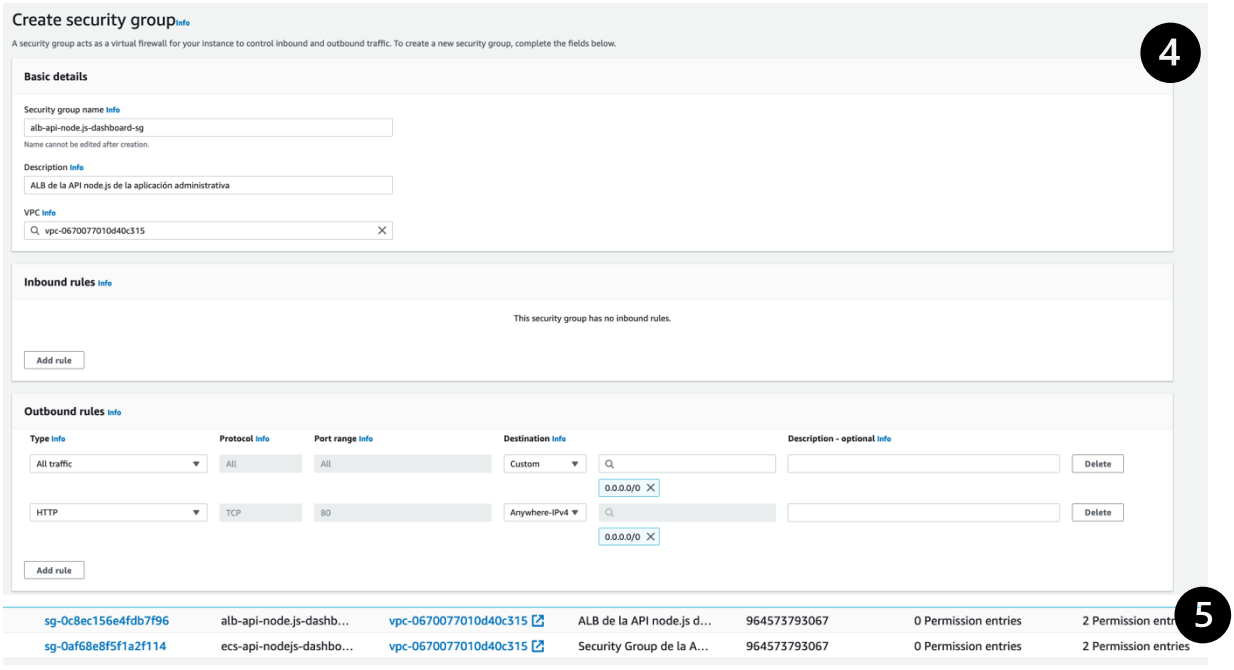
Fargate:
A continuación, procedemos a crear un clúster de Fargate:



3.2.1. *Security Groups*

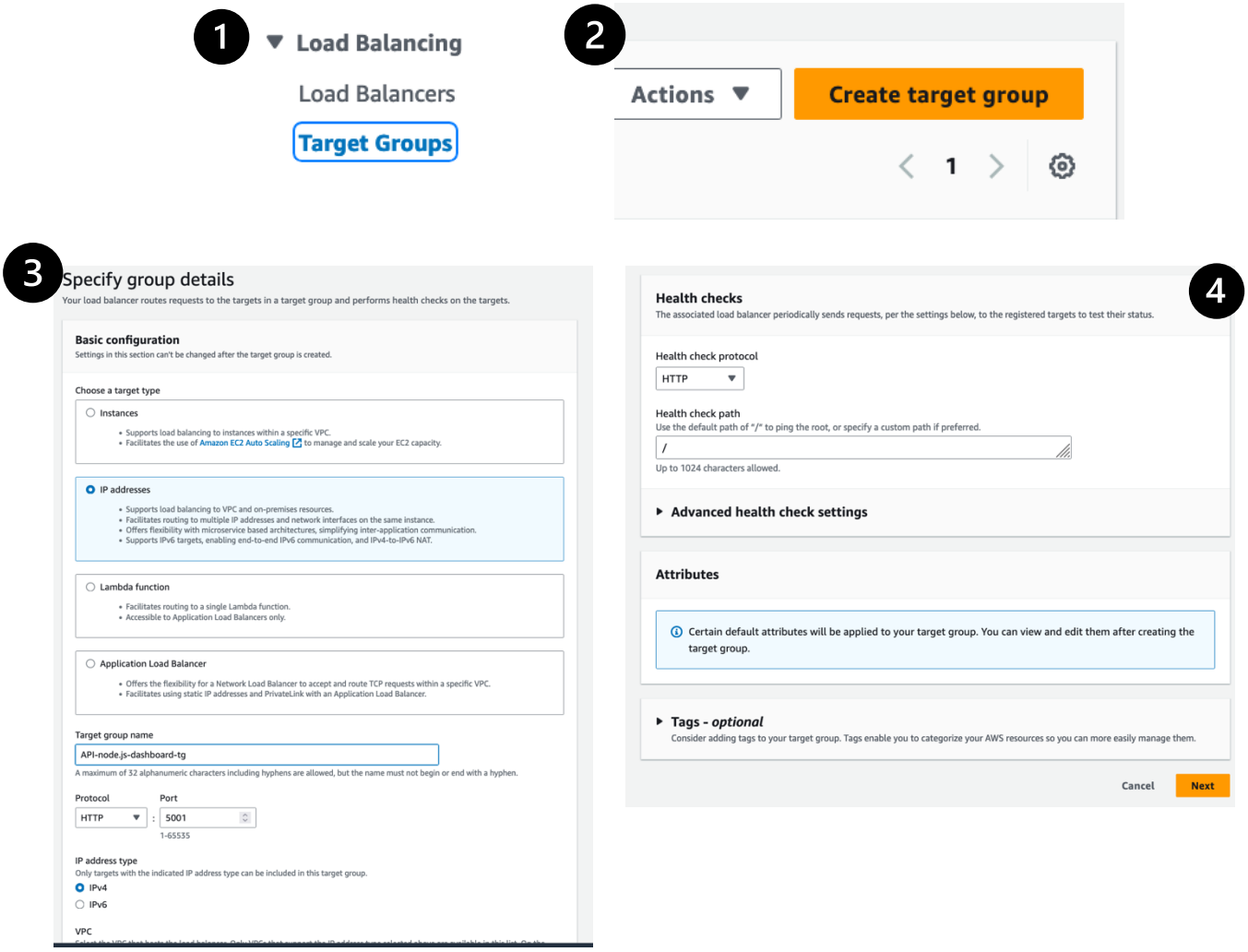
Después de crear el clúster de *Fargate*, procedemos a crear los demás recursos, en este caso, los *Security Groups* (vienen a ser los Firewalls de AWS). Para llevar a cabo la siguiente instalación, nos situaremos en el servicio ECS de Amazon.





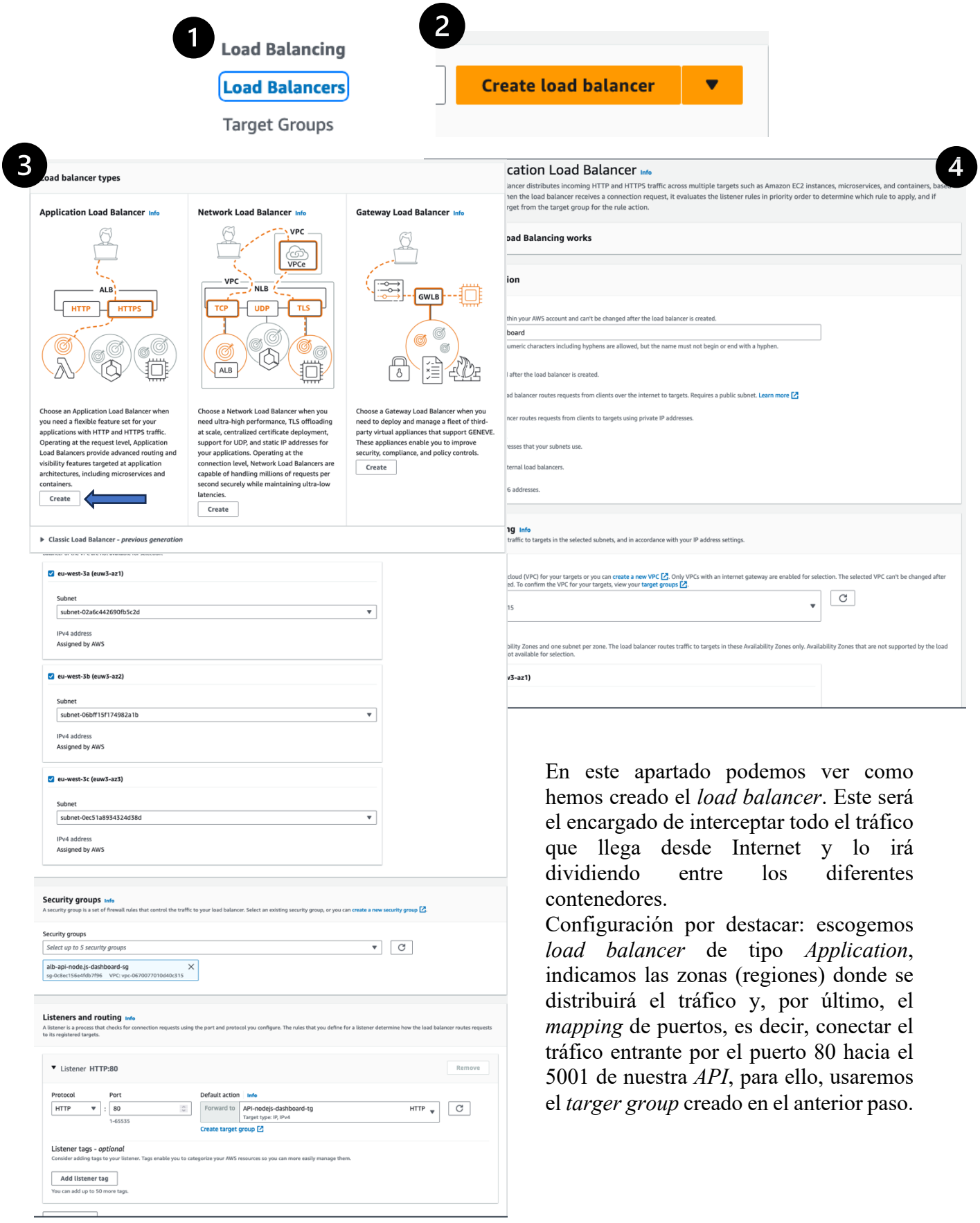
Como podemos observar en estas capturas, hemos creado dos *firewalls*, uno para el *Load balancer* y el otro es para el entorno de las réplicas de contenedores.

3.2.2. *Target Groups*



En esta parte hemos creado un *target group*, es decir, es el componente utilizado para enrutar el tráfico hacia las instancias o contenedores. En este caso lo creamos vacío, ya que *Fargate* será el encargado de crear dicho contenido.
En las capturas mostradas, configuramos: el *target group* de tipo *IP addresses*, protocolo y el puerto, y finalmente la dirección del *Health check* (un método para saber si el contenedor está vivo)

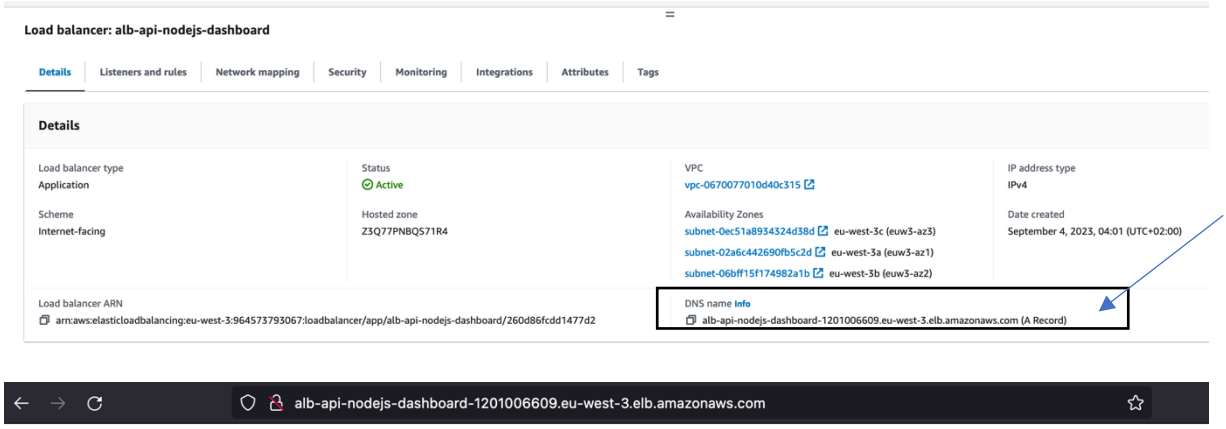
3.2.3. *Load Balancer*



En este apartado podemos ver como hemos creado el *load balancer*. Este será el encargado de interceptar todo el tráfico que llega desde Internet y lo irá dividiendo entre los diferentes contenedores.

Configuración por destacar: escogemos *load balancer* de tipo *Application*, indicamos las zonas (regiones) donde se distribuirá el tráfico y, por último, el *mapping* de puertos, es decir, conectar el tráfico entrante por el puerto 80 hacia el 5001 de nuestra *API*, para ello, usaremos el *targer group* creado en el anterior paso.

Lo más importante antes de continuar, es verificar el correcto funcionamiento del *load balancer*. Para ello, accedemos a las credenciales de este mismo y recuperamos el parámetro *DNS Name*.
Una vez recuperado el *DNS* del l*oad balancer*, mediante el navegado intentamos acceder.

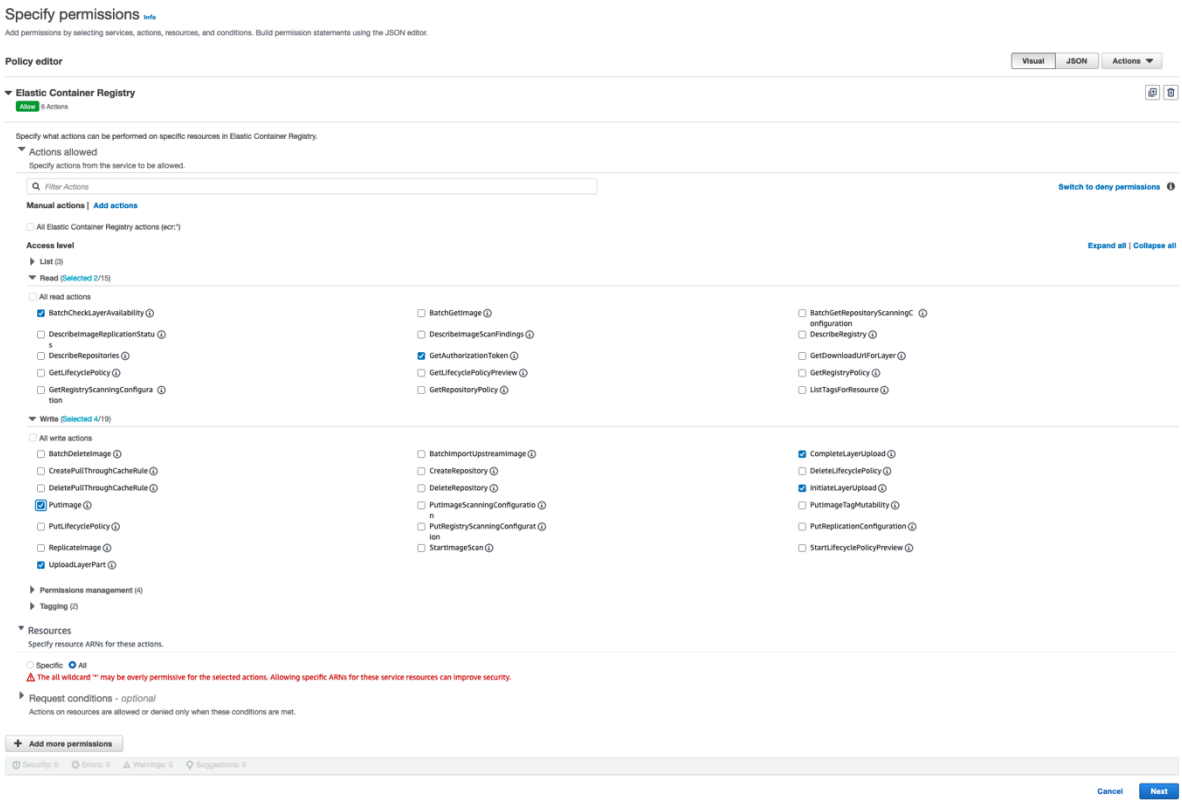


503 Service Temporarily Unavailable

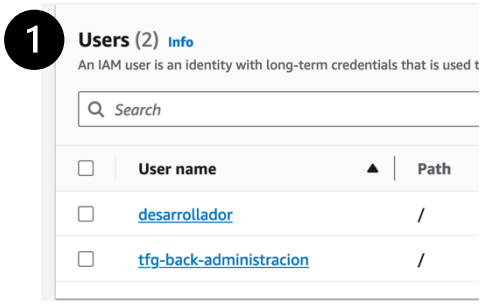
Como podemos notar, al acceder vía navegador al *DNS* del *load balancer,* obtenemos un mensaje del servidor con un error de código 503, lo que significa que este funciona, solo que no encuentra ningún recurso.

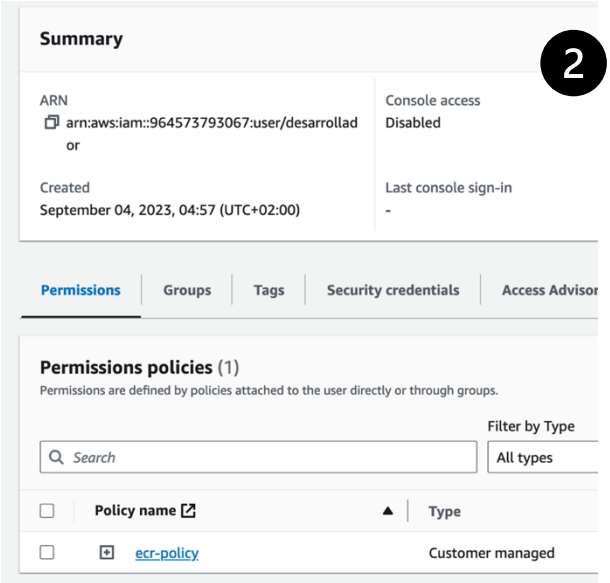
3.2.4. *Amazon Elastic Container Registery (ECR)*

Una vez ya tenemos el *load balancer* apuntando al clúster de *Fargate*, para crear los contenedores que el servicio de *Fargate* irá escalando según la carga, necesitamos la imagen base (*Docker Image*).
Amazon nos ofrece el servicio ECR [44]. Es un servicio de registro de imágenes de contenedor administrado por AWS que es seguro, escalable y fiable.



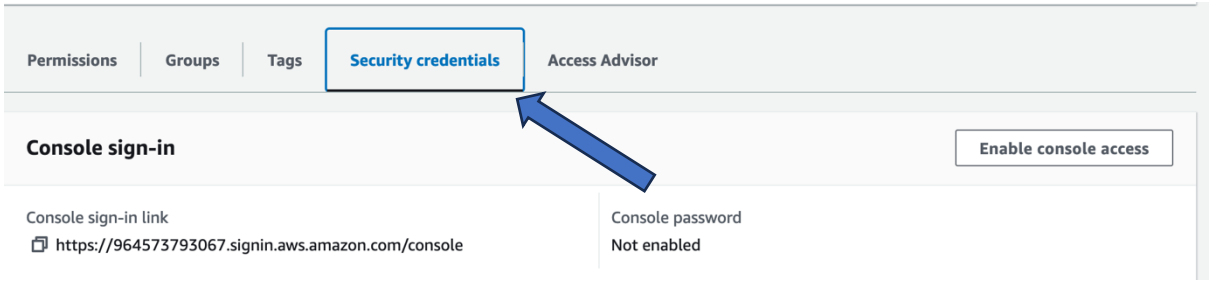
En esta captura, podemos observar cómo creamos una nueva política que nos permite subir una imagen al servicio ECR mediante el servicio administración de identidades (IAM). Ahora tenemos que añadir un usuario que se encargará se subir imágenes a la consola de *AWS*:

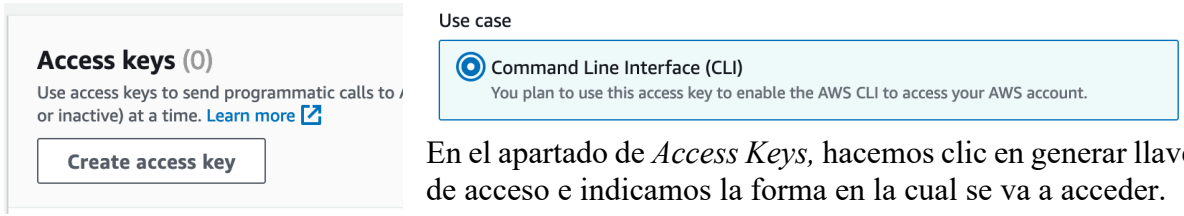




Como podemos notar, este nuevo usuario tiene permisos para subir una imagen al repositorio ECR.

Para iniciar sesión con este usuario en la máquina que contiene la imagen a subir, debe tener unas llaves de acceso, para ello seleccionamos este usuario:





En el apartado de *Access Keys,* hacemos clic en generar llaves de acceso e indicamos la forma en la cual se va a acceder.

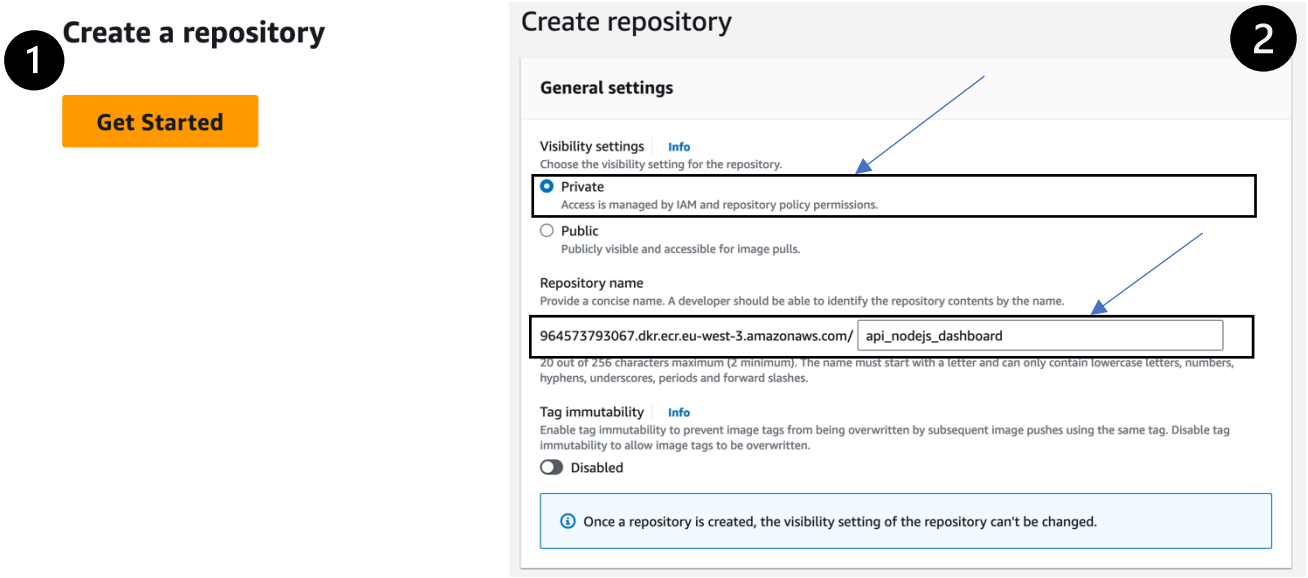
```
brew install awscli
```

Nota: Es importante tener la *AWS CLI* instalada, en este caso usamos el gestor de paquetes de Mac (*brew*).

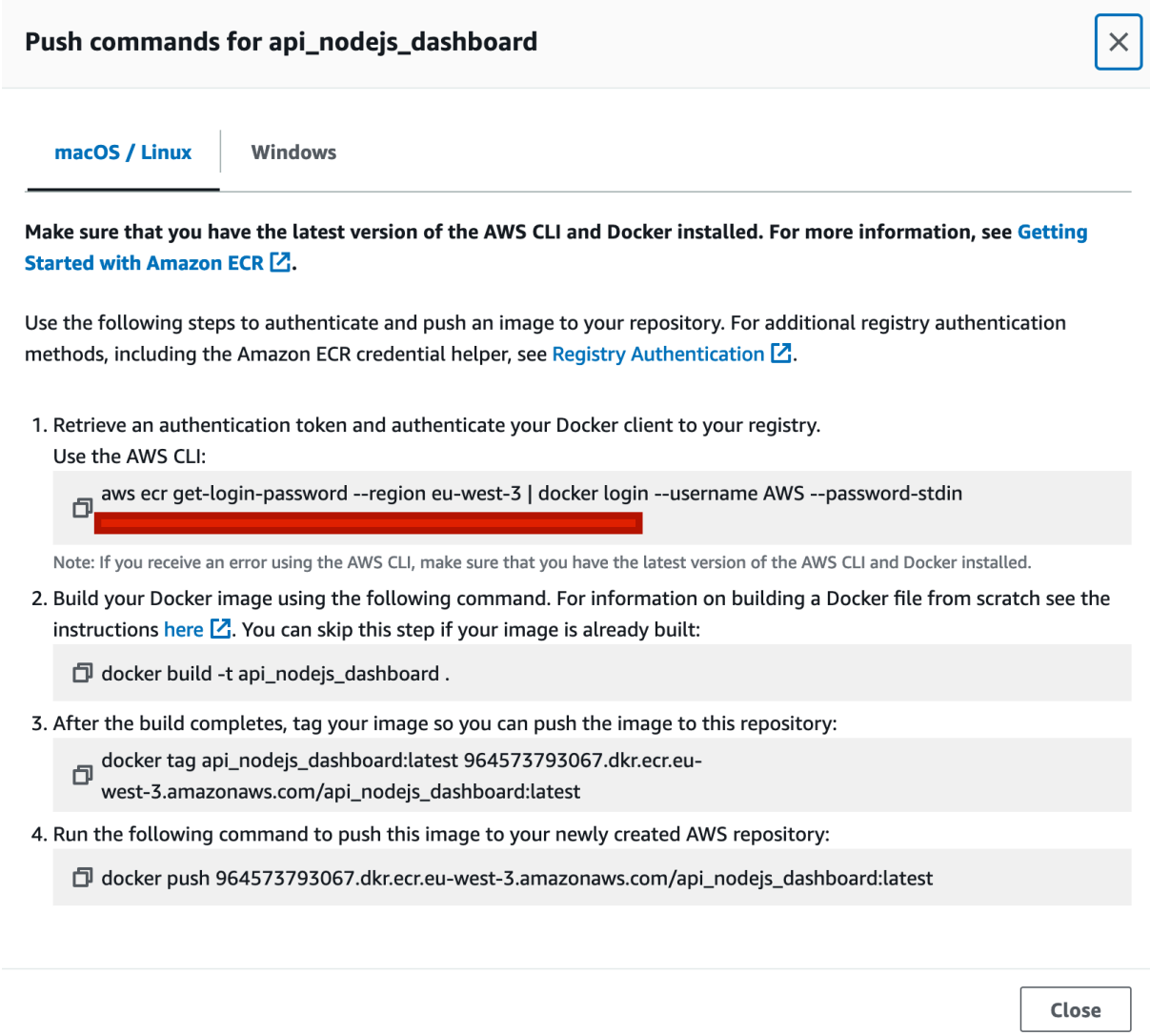
```
youneskabiri@Youness-MacBook-Pro ~ % aws configure
AWS Access Key ID [None]: AKIA6BFIUBMVYKG4X7KW
AWS Secret Access Key [None]: [redacted]
Default region name [None]: eu-north-1
Default output format [None]:
youneskabiri@Youness-MacBook-Pro ~ %
```

***Figura 54**. Inicio sesión AWS CLI.*

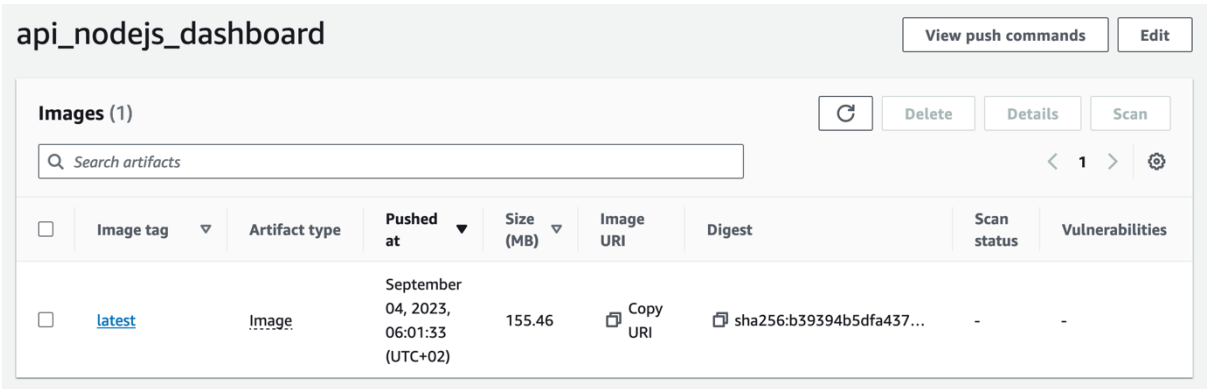
Después de iniciar sesión, ya podemos crear el repositorio donde se subirá la imagen.



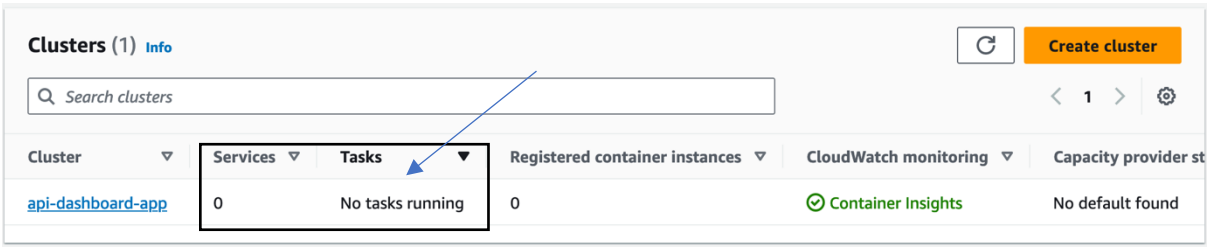
Siguiendo los pasos que nos dice la consola de AWS, procedemos a subir la imagen.



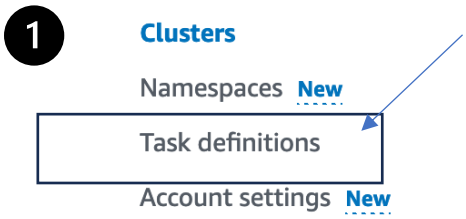
Con solo ejecutar las comandas mostradas en la captura, nuestra imagen ya se habrá subido a AWS.

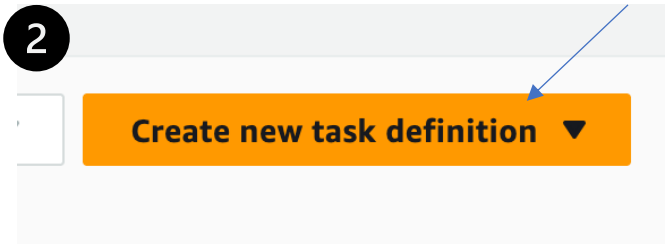


3.2.5. Asignar recursos a Fargate

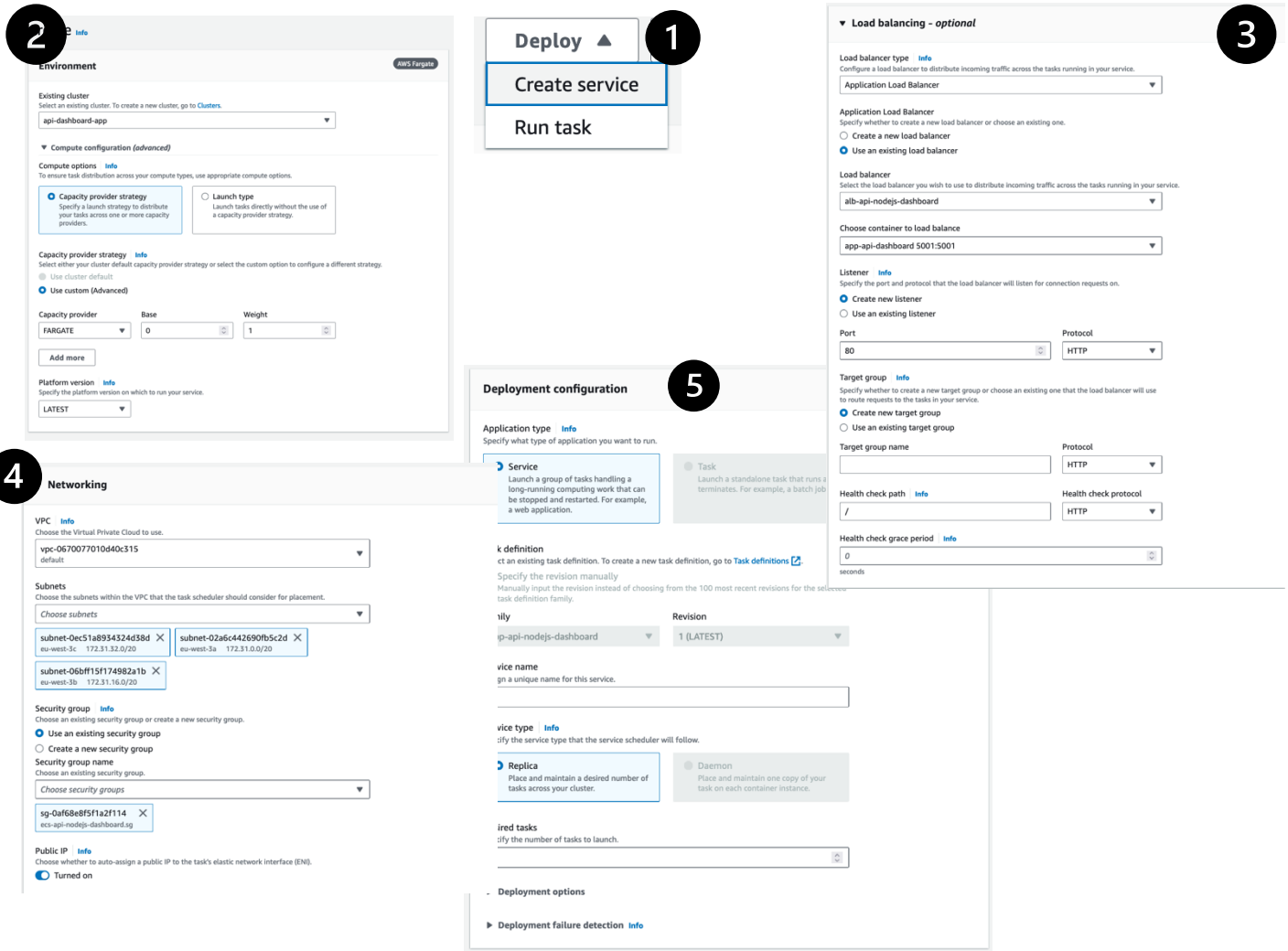


Si volvemos al clúster de *Fargate*, poder ver que nos dice que no hay tareas, es decir, que este servicio no tiene ningún conjunto de contenedores ejecutando, para ello debemos crear esta tarea.





Una vez creado los recursos, tenemos que desligar estos



7. Conclusiones

El hecho de aceptar un proyecto de esta magnitud nos ha ayudado a aprender muchas cosas, entre ellas adquirir la capacidad de interpretar las necesidades de nuestro cliente y transformar sus complicaciones en soluciones reales, gracias a la combinación perfecta de tecnologías y herramientas. No solo nos ha impulsado a reforzar y poner en práctica todo lo aprendido durante los últimos años en la carrera, sino también aprender conceptos y tecnologías nuevas.

A pesar de obtener el resultado esperado y satisfacer las necesidades del cliente, para llegar a este resultado, hemos tenido que enfrontar varios obstáculos, pero estos no nos han detenido, y solo han servido para encontrar otras salidas y ser más organizados. Gracias a este proyecto, nuestra carrera profesional se ha alzado, ya que nos ha brindado más confianza para aceptar futuros proyectos de la misma magnitud o incluso mayores.

Como futura funcionalidad, hay un conjunto de ideas que la empresa nos ha aconsejado después de utilizar el sistema por un tiempo, como por ejemplo, la funcionalidad de fichar a los trabajadores de las diferentes obras, contabilizar los gastos de las dietas o la gestión de los pagos del cliente. A nivel de desarrollo, nos gustaría automatizar el proceso del despliegue, es decir, incorporar el tema de CI-CD. También aprovechar más las ventajas que nos ofrece el *Cloud computing* mediante funciones *Serverless* y más.

8. Referencias

[1]. Tekla. (2022, 17 marzo). ¿Qué es el desarrollo web? [Todo lo que necesitas saber]. *TEKLA*. https://tekla.io/blog/que-es-desarrollo-web/

[2]. Laoyan, S. (2022, 29 septiembre). Qué es la metodología Waterfall y cuándo utilizarla [2022] • Asana. *Asana*. https://asana.com/es/resources/waterfall-project-management-methodology

[3]. Mendoza, M. L. (2023, 13 abril). Qué es un lenguaje de programación. *OpenWebinars.net*. https://openwebinars.net/blog/que-es-un-lenguaje-de-programacion/

[4]. Tablado, F. (2020). Bases de datos. ¿Qué son? tipos y ejemplos. *Ayuda Ley Protección Datos*. https://ayudaleyprotecciondatos.es/bases-de-datos/#Que_es_una_base_de_datos_Como_se_define

[5]. *IBM documentation*. (s. f.). https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions

[6]. Maldeadora. (2017). Bases de datos: qué tipos existen y cómo funcionan. *Platzi*. https://platzi.com/blog/bases-de-datos-que-son-que-tipos-existen/

[7]. Carrero, L. (2023, 23 mayo). Bases de datos NoSQL: características y tipos | Stackscale. *Stackscale*. https://www.stackscale.com/es/blog/bases-de-datos-nosql/

[8]. MongoDB. (s. f.). *NoSQL vs SQL Databases*. https://www.mongodb.com/nosql-explained/nosql-vs-sql

[9]. *Bases de datos NoSQL: qué son y cuáles son sus ventajas*. (2021, 1 octubre). UNIR México. https://mexico.unir.net/ingenieria/noticias/bases-de-datos-nosql/

[10]. Gonzalez, S. (s. f.). *¿Qué es la experiencia de usuario?* https://www.cyberclick.es/que-es/experiencia-de-usuario

[11]. Northware. (2022). Requerimientos en el desarrollo de software y aplicaciones. *Northware*. https://www.northware.mx/blog/requerimientos-en-el-desarrollo-de-software-y-aplicaciones/

[12]. Powell, Z. (2023). Angular vs React: una comparación en profundidad. *Kinsta®*. https://kinsta.com/es/blog/angular-vs-react/

[13]. Team, K. (2023, 21 abril). ¿Qué es el Patrón de Arquitectura MVVM? *KeepCoding Bootcamps*. https://keepcoding.io/blog/que-es-el-patron-de-arquitectura-mvvm/

[14]. Vidal, M. (2022, 2 noviembre). *PWA: Qué son y cómo funcionan las Progressive Web Apps*. Thinking for Innovation. https://www.iebschool.com/blog/progressive-web-apps-analitica-usabilidad/

[15]. *Qué es el DOM*. (s. f.). DesarrolloWeb.com. https://desarrolloweb.com/articulos/que-es-el-dom.html

[16]. Glover. (2022). Reducing the size of your react applications with tree shaking and code splitting. *DEV Community*. https://dev.to/_itsglover/reducing-the-size-of-your-react-applications-with-tree-shaking-and-code-splitting-1650

[17]. *TypeORM en Nest Framework*. (s. f.). DesarrolloWeb.com. https://desarrolloweb.com/articulos/typeorm-nest-framework

[18]. Oleaga, J. (2020, 18 junio). Los servicios en la «nube» aumentan durante la crisis del Covid-19. *Diario ABC*. https://www.abc.es/tecnologia/informatica/soluciones/abci-servicios-nube-aumentan-durante-crisis-covid-19-202006181030_noticia.html

[19]. *Docker: a DevOps tool*. (2019, 7 octubre). inLab FIB. https://inlab.fib.upc.edu/ca/blog/docker-devops-tool
[20]. Nuñez, E. A. (2023, 21 abril). Docker, qué es y sus principales características. *OpenWebinars.net*. https://openwebinars.net/blog/docker-que-es-sus-principales-caracteristicas/
[21]. Team, K. (2023a, enero 12). ¿Qué es un contenedor en docker? | KeepCoding Bootcamps. *KeepCoding Bootcamps*. https://keepcoding.io/blog/que-es-un-contenedor-en-docker/
[22]. Zanini, A. (2023, 24 julio). *Using helmet in Node.js to secure your application - LogRocket blog*. LogRocket Blog. https://blog.logrocket.com/using-helmet-node-js-secure-application/
[23]. KeepCoding, R. (2023, 6 enero). ¿Qué es JSON Web Token? | KeepCoding Bootcamps. *KeepCoding Bootcamps*. https://keepcoding.io/blog/que-es-json-web-token/
[24]. Sosa, A. L. (2021, 10 diciembre). Middleware en Express JS - Aarón López Sosa - Medium. *Medium*. https://medium.com/@aarnlpezsosa/middleware-en-express-js-5ef947d668b
[25]. MikeRayMSFT. (2023, 28 febrero). *Binary Large Object (BLOB) Data (SQL Server) - SQL Server*. Microsoft Learn. https://learn.microsoft.com/en-us/sql/relational-databases/blob/binary-large-object-blob-data-sql-server?view=sql-server-ver16
[26]. *How to work with BLOB in MySQL database hosted on Alibaba Cloud*. (s. f.). Alibaba Cloud Community. https://www.alibabacloud.com/blog/how-to-work-with-blob-in-mysql-database-hosted-on-alibaba-cloud_594270
[27]. *¿Qué es YAML?* (s. f.). https://www.redhat.com/es/topics/automation/what-is-yaml
[28]. *ElasticSearch: Motor de búsqueda y analítica distribuido oficial | Elastic*. (s. f.). Elastic. https://www.elastic.co/es/elasticsearch/
[29]. *¿Qué es grafana?* (s. f.). https://www.redhat.com/es/topics/data-services/what-is-grafana
[30]. *«Docker Compose Overview»*. (2023, 30 agosto). Docker Documentation. https://docs.docker.com/compose/
[31]. *Postman API Platform*. (s. f.). https://www.postman.com/
[32]. *Amazon DocumentDB*. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/documentdb/
[33]. Angular, A. A. A. 2.-. I. A. L. S. E. (2017, 20 diciembre). *Introducción a Angular (2) actualizada: Módulo, Componente, Template y metadatos*. Enrique Oriol. http://blog.enriqueoriol.com/2017/03/introduccion-angular-modulo-y-componente.html
[34]. *Angular*. (s. f.). https://docs.angular.lat/guide/architecture
[35]. Digital. (2022, 4 octubre). *Cómo usar testing en Angular Con Jasmine y Karma*. DIGITAL55. https://digital55.com/blog/como-usar-testing-angular-jasmine-karma/
[36]. Ledo, A. M. (2021, 13 diciembre). Configurar TSLINT en cualquier proyecto de Typescript. *Medium*. https://mugan86.medium.com/configurar-tslint-en-cualquier-proyecto-de-typescript-f2f09bf33c43
[37]. *Angular*. (s. f.-b). https://angular.io/guide/lazy-loading-ngmodules
[38]. KeepCoding, R. (2022, 30 noviembre). ¿Qué es TypeScript? | KeepCoding Bootcamps. *KeepCoding Bootcamps*. https://keepcoding.io/blog/typescript/

[39]. *El sector 'start-up' español aumenta el crecimiento y las ventas online, pero reduce rentabilidad y actividad internacional*. (s. f.). comunicacion. https://www.caixabank.com/comunicacion/noticia/el-sector-start-up-espanol-aumenta-el-crecimiento-y-las-ventas-online-pero-reduce-rentabilidad-y-actividad-internacional_es.html?id=43610#

[40]. *CONVOCATORIA DE AYUDAS DESTINADAS a LA DIGITALIZACIÓN DE EMPRESAS DEL SEGMENTO I (ENTRE 10 y MENOS DE 50 EMPLEADOS), DENTRO DEL PROGRAMA KIT DIGITAL | Sede*. (s. f.). https://sede.red.gob.es/es/procedimientos/convocatoria-de-ayudas-destinadas-la-digitalizacion-de-empresas-del-segmento-i-entre

[41]. Benito, M. (2022, 18 marzo). *Qué es Node.JS y cuáles son las ventajas de usar esta tecnología*. FP Online. https://fp.uoc.fje.edu/blog/que-es-node-js-y-cuales-son-las-ventajas-de-usar-esta-tecnologia/

[42]. *¿Qué es Amazon Elastic Container Service? - Amazon Elastic Container Service*. (s. f.). https://docs.aws.amazon.com/es_es/AmazonECS/latest/developerguide/Welcome.html

[43]. *¿Qué es AWS Fargate? - Amazon ECS*. (s. f.). https://docs.aws.amazon.com/es_es/AmazonECS/latest/userguide/what-is-fargate.html

[44]. *¿Qué es Amazon Elastic Container Registry? - Amazon ECR*. (s. f.). https://docs.aws.amazon.com/es_es/AmazonECR/latest/userguide/what-is-ecr.html

[45]. Edix, R. (2022, 26 julio). *Framework: qué es, para qué sirve y algunos ejemplos*. Edix España. https://www.edix.com/es/instituto/framework/