ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Grado en Ingeniería del Software

Red social de sitios de interés público

Social network of places of public interest

Realizado por

José Luis Soto-Aranaz Muñoz

Tutorizado por

Eduardo Guzmán de los Riscos

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE 2022

Fecha defensa: septiembre de 2022

Resumen

Hoy en día existen varias plataformas que cubren las necesidades de socialización entre personas gracias a distintas redes sociales de carácter general. La mayoría de ellas son específicas, para subir videos (Youtube), para anunciarse y encontrar personas con las que has perdido el contacto (Facebook), enviar pequeños mensajes de información (Twitter), enviar imágenes o pequeños videos (Instagram), etc.

La idea de este TFG trata de cubrir la necesidad de una red social enfocada a la cultura, paisajes y distintas localizaciones que sean de relevancia para otras personas. En esta red social se podrán subir imágenes de nuestro entorno, con su correspondiente localización para que otras personas puedan disfrutar de las vistas y lugares que no sabían que existían.

Palabras clave: **Red social, Lugares de interés, Nodejs, Angular, Mongo DB**

Abstract

Today there are several platforms that cover the needs of socialization between people through different social networks of a general nature. Most of them are specific, to upload videos (Youtube), to advertise and find people with whom they have lost contact (Facebook), send small information messages (Twitter), send images or small videos (Instagram) etc.

The Idea of this TFG tries to cover the need for a social network focused on culture, landscapes and different locations that are relevant to other people. In this social network, Images of our environment can be uploaded, with their corresponding locations that other people can enjoy the views and places that they did not know existed.

Keywords: **Social network, Places of Interest, Nodejs, Angular, Mongo DB**

Índice

Resumen 1

Abstract 1

Índice 1

Introducción 5

1.1 Motivación 5

1.1.1 Ventajas de la aplicación frente a otras redes sociales 5

1.2 Objetivos 6

1.3 Estructura de la memoria 12

1.4 Tecnologías y herramientas utilizadas 6

1.4.1 Entorno de desarrollo - Atom 7

1.4.2 AJAX 7

1.4.3 Node.js 8

1.4.4 NestJS 8

1.4.5 Express.js 8

1.4.6 MongoDB 9

1.4.7 JSON 9

1.4.8 Angular 9

1.4.9 Git 10

1.4.10 Git Bash 10

1.4.11 Less 10

1.4.12 DigitalOcean 11

1.4.13 Heroku 11

1.4.14 Otros 11

Arquitectura 13
2.1. Arquitectura del proyecto 13
2.1.2 Diagramas de secuencia 15
2.2. Arquitectura de los entornos de trabajo (frameworks) 25
2.2.1 Arquitectura en NestJS 25
2.2.2 Arquitectura en Angular 25
2.2.3 Arquitectura en MongoDB 26
Análisis 27
3.1. Catálogo de actores 27
3.2. Catálogo de requisitos 28
3.2.1 Requisitos funcionales 28
3.2.1.1 RF01 Gestión de usuarios 28
3.2.1.2 RF02 Gestión de publicaciones 29
3.2.1.3 RF03 Gestión de comentarios 29
3.2.1.4 RF04 Gestión de valoraciones 30
3.2.1.5 RF05 Gestión de Categorías 30
3.2.1.6 RF06 Gestión de Etiquetas 30
3.2.2 Requisitos no funcionales 31
3.2.2 Requisitos de información 32
3.3 Casos de uso 32
Implementación 55
4.1. Metodología de desarrollo 55
4.2. Implementación Servidor 57
4.2.1. Base de datos MongoDB 57
4.2.2. API 58
4.2.3. DigitalOcean 67
4.2.4. Postman 70
4.3. Implementación Cliente 73
4.3.1. Angular y Angular Material 73
4.4. Subida de archivos a la nube 88
4.4.1. GitHub 88

4.4.2. Heroku 91

Conclusiones y mejoras futuras 97

5.1. Conclusiones 97

5.2. Problemáticas en la ejecución del proyecto 98

5.3. Mejoras futuras 100

Referencias 101

Manual de Instalación 109

Requerimientos: 109

Instalación: 110

1

Introducción

1.1 Motivación

La motivación de este proyecto surge a través de la necesidad del ser humano por comunicarse con otras personas. Ninguna de las plataformas actuales de comunicación abarca necesidades como contribuir en la información de sitios culturales o paisajes y localizaciones que pueden ser de especial interés para distintos miembros de redes sociales. Es por ello por lo que se ha decidido desarrollar una red social que se ocupe de esta temática.

1.1.1 Ventajas de la aplicación frente a otras redes sociales

Twitter es una red social donde se pueden añadir imágenes y pequeños comentarios. La principal diferencia con esta aplicación es que aquí se puede disponer de la localización dónde la foto ha sido tomada para que los usuarios puedan saber su ubicación e ir a ver su contenido en cualquier momento. Algo parecido ocurre en Instagram, otra red social en la que se pueden subir imágenes en cada perfil de usuario, aunque no son públicas para que todo el mundo pueda verlas, sino que tienes que entrar en el perfil del usuario concreto al que quieres ver sus imágenes. Esta aplicación garantiza que cualquier usuario pueda ver cualquier foto que se suba a esta web sin necesidad de registro o entrar a ningún perfil.

1.2 Objetivos

Este proyecto se centra en el desarrollo de una aplicación web capaz de transmitir a múltiples usuarios la localización exacta de sitios de interés cultural, así como una descripción de los mismos con imágenes y texto.
Cualquier usuario podrá ser miembro de esta comunidad a través de un registro y subir las localizaciones que sean pertinentes para él. Estas localizaciones podrán ser filtradas por el resto de usuarios de la red a través de distintos medios como pueden ser una serie de etiquetas o categorías.

Los usuarios podrán hacer valoraciones de estos lugares de interés, así como comentarios en los mismos, ayudando a que los sitios más destacados puedan ser más vistos por otros miembros de esta comunidad.
Cada usuario tendrá su listado de lugares de interés, tanto los que él a dado a conocer, como los que quiera guardar en un listado de sitios favoritos para poder ir a visitarlos en cualquier momento.

1.3 Tecnologías y herramientas utilizadas

El desarrollo de la aplicación se hará con las últimas tecnologías en materia web, utilizando Node.js. Se usará AJAX como conjunto de técnicas para el desarrollo de la aplicación. En la parte servidora se realizará una API desarrollada con el framework NestJS, con base del framework Express atacando a una base de datos implementada en MongoDB. El cliente intercambiará información con este servidor mediante distintos métodos sobre HTTP y estará desarrollado con el framework Angular con una tecnología basada en SPA (single page application). Como cualquier web se usará HTML5 y CSS3 para el desarrollo de la parte cliente. Para facilitar la carga del CSS se usará Less, que lo compilará y comprimirá. Además, se hará uso de distintas librerías para ayudar al diseño de la aplicación, como pueden ser Boostrap o Font Awesome. Para la comunicación entre el servidor y el cliente se usará el formato de texto JSON.

Para tener los archivos en la nube y tener un control de versiones se usará **GIT**, adicionalmente se utilizará la consola **Git Bash** para poder trabajar con el protocolo HTTP de forma segura a través de SSH. Como alojamiento de los archivos se usará **Heroku**, exceptuando los archivos que suba el usuario que se almacenarán en **DigitalOcean**.

1.3.1 Entorno de desarrollo - Atom

Atom es un editor de texto de código abierto desarrollado por GitHub. Está disponible en varios sistemas operativos como Linux, Windows y Mac. Fue desarrollado usando tecnologías web y sus funcionalidades pueden ser ampliadas con distintos conectores desarrollados con Node.js y se instalan con facilidad a través de su gestor de paquetes. Mediante las posibilidades que ofrece se puede personalizar de una forma muy cómoda, eligiendo las características que más convengan en cada momento, y con un rendimiento muy alto.

1.3.2 AJAX

AJAX es el acrónimo de *Asynchronous JavaScript And XML.* Esta técnica de desarrollo se usa en la web para crear aplicaciones que tienen la facultad de ejecutarse en el navegador del cliente, pero comunicándose con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas web sin necesidad de volver a cargarlas gracias a una comunicación asíncrona, lo que aumenta la interactividad, rapidez y el uso en las aplicaciones. Esta tecnología es asíncrona, en el sentido de que los datos que se le piden al servidor a través de una petición se cargan en segundo plano y no modifican la visualización ni el comportamiento de la página. Normalmente tras recibir los datos se hace una actualización de la página en la aplicación para mostrar los datos. La mayoría de las veces el lenguaje que se usa es JavaScript en la parte del cliente, un lenguaje interpretado con mucho uso hoy en día, mientras que el acceso a los datos se realiza mediante XMLHttpRequest, un objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML, de hecho, para esta aplicación el lenguaje a utilizar será JSON.

1.3.3 Node.js

Node.js es una tecnología relativamente nueva comparada con otras tecnologías web. Está basado en el lenguaje de programación JavaScript, de hecho, hasta que se creó Node.js este lenguaje tan sólo podía usarse del lado del cliente. Gracias a Node.js se empezó a usar JavaScript del lado del servidor.

Node.js es un entorno multiplataforma y de código libre. Se creó para programas de red altamente escalables, como pueden ser los servidores web.

Node.js es sumamente concurrente, ya que funciona con un modelo de un único hilo de ejecución, usando cientos de miles de entradas y salidas asíncronas que se ejecutan concurrentemente. Una peculiaridad es que todas las operaciones de entradas y salidas deben tener una función callback que puede devolver algo.

1.3.4 NestJS

NestJS es un framework progresivo de Node.js para la creación de aplicaciones eficientes, confiables y escalables del lado del servidor. Es completamente compatible con TypeScript aunque permite también la codificación con JavaScript puro. Combina elementos de la programación orientada a objetos, programación funcional y programación reactiva funcional.

Usa por defecto el framework Express.js para diversas funcionalidades, y está inspirado en Angular, así que comparte multitud de conceptos con este framework que se basa en decoradores. Su arquitectura modular es muy flexible, permitiendo usar múltiples bibliotecas existentes que hacen la programación más sencilla, siendo muy versátil, y altamente escalable.

1.3.5 Express.js

Express.js es un marco de aplicación para servidores web específico para Node.js. Es un software gratuito y de código abierto bajo la licencia MIT, diseñado para crear aplicaciones web y APIs. Se le ha llamado el estándar de facto marco servidor para Node.js. Fue creado por

TJ Holowaychuk, que lo describió como un framework mínimo, pero con muchísimas características disponibles como complemento.
Express es un componente de backend de pilas de desarrollo muy populares como la pila MEAN (Mongo, Express, Angular, Node).

1.3.6 MongoDB

MongoDB, del inglés *hu**mongo**us* que significa enorme, es un sistema de base de datos **NoSQL** de código libre. MongoDB no guarda los datos en tablas como en la mayoría de bases de datos que son relacionales, sino que las almacena es unas estructuras de datos similares a JSON llamadas **BSON.** Esto facilita la incorporación de los datos en algunas aplicaciones, consiguiendo que sea más sencilla y veloz. Este tipo de estructura es muy útil en estructuras de contenido gigantesco, de ahí el nombre.

1.3.7 JSON

JSON es el acrónimo de notación de objeto de JavaScript en inglés. JSON se usa para enviar datos en formato de texto fácilmente legibles. Su notación es una estructura con pares clave valor con dos puntos en medio para separarlas. Mediante llaves se puede generar una estructura simple como si fuera un array en cualquier lenguaje de programación, pudiendo generar tantos grupos como se quiera. Al inicio de su creación estaba unido al lenguaje JavaScript, aunque desde 2019 se le considera totalmente independiente. Es mucho más simple que XML, por lo que hoy en día es muchísimo más utilizado para el intercambio de datos haciendo que XML baje a la segunda posición.

1.3.8 Angular

Angular es un framework opensource desarrollado por Google para facilitar la creación y programación de aplicaciones web de una sola página, conocidas como SPA (Single Page

Application). Usa un patrón Modelo-Vista-Controlador (MVC) que mantiene todo más ordenado haciendo que el código sea más veloz, menos repetitivo posibilitando modificaciones y actualizaciones más rápidamente. Como ventaja principal se puede decir que es altamente escalable y está basado en el estándar de componentes web, además de permitir crear nuevas etiquetas HTML personalizadas que se pueden reutilizar.

1.3.9 Git

Git es un software de control de versiones diseñado por Linus Torvalds. Fue ideado para mejorar la eficiencia y que fuera más confiable y compatible el mantenimiento de versiones de cualquier aplicación, sobre todo cuando tienen un gran volumen de. Su finalidad es llevar un registro de todos los cambios de cualquier archivo que incluye las fechas de cambio, las diferencias entre las versiones, anotaciones, así como coordinar el trabajo que varias personas realizan sobre archivos compartidos en un repositorio de código. Git es un software libre distribuible bajo los términos de la versión 2 de la Licencia Pública General de GNU.
Sus puntos fuertes son el apoyo al desarrollo no lineal por lo que tiene una alta velocidad en la gestión de ramas y el manejo de versiones, gestión distribuida que hace que los cambios se propaguen entre los repositorios locales, publicación de almacenes mediante HTTP o FTP, gestión eficiente de proyectos grandes, compatibilidad con diversos IDEs, etc. Además, plantea una gran libertad en la forma de trabajar en torno a un proyecto, aunque los miembros del mismo deben acordar el flujo de trabajo que van a seguir.

1.3.10 Git Bash

Git Bash es la línea de comandos de Git para Windows que permite lanzar comandos de Linux básicos para múltiples tareas. Es la opción más utilizada para usar Git en Windows.

1.3.11 Less

Less es un dinámico lenguaje de estilo de código abierto que puede ser compilado en hojas de estilo en cascada (CSS) y ejecutarse en el lado del cliente o en el lado del servidor. La ventaja

que tiene con respecto al CSS tradicional es que permite que se definan variables, incrustar propiedades de una clase dentro de otra, realizar anidamientos de código, así como utilizar funciones y operaciones.

1.3.12 DigitalOcean

DigitalOcean es un proveedor de servidores virtuales privados. En esta nube se almacenan las imágenes y o archivos que los usuarios quieren guardar en la aplicación.

1.3.13 Heroku

Heroku es una de las primeras plataformas como servicio (PaaS) de computación en la nube, propiedad de Salesforce.com, que soporta distintos lenguajes de programación. Para este proyecto el lenguaje de los archivos es *Typescript*.

1.3.14 Otros

Aunque el lector probablemente ya conozca algunas de estas tecnologías nunca viene mal hacer un repaso de éstas.

HTML es el lenguaje de marcado que se usa hoy en día para el desarrollo y creación de páginas web. Se compone de una serie de etiquetas que se incluyen dentro del lenguaje.

CSS es el acrónimo inglés de las *Hojas de estilo en cascada*. Es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en HTML.

Bootstrap es una biblioteca multiplataforma o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web. Sus plantillas de diseño contienen distintos elementos para facilitar el diseño de múltiples elementos HTML.

Font Awesome es un framework de iconos vectoriales y estilos CSS utilizado para sustituir imágenes de iconos comunes por gráficos vectoriales convertidos en fuentes.

Tras el acceso del usuario a la aplicación cliente, el servidor de aplicación (Web Server) realiza peticiones API (Application Server), comunicándose así con el servidor de la base de datos (DB Server), que a su vez accede a la base de datos en la nube (Database) y ejecuta las sentencias que correspondan.

1.4 Estructura de la memoria

En este primer capítulo, la Introducción, se comentan los objetivos, la motivación del proyecto y la solución adoptada para abordar el problema a resolver. Incluye los recursos software, así como las tecnologías que se han utilizado para la realización de la aplicación desarrollada en este proyecto.

El segundo capítulo trata sobre la arquitectura que se ha decidido usar para realizar el proyecto, en los frameworks y en la base de datos. Se adjuntan varios diagramas de secuencia que modelan los procesos que lanza el usuario al interactuar con la aplicación.

El tercer capítulo aborda la funcionalidad requerida por el cliente, desglosando los requisitos funcionales, no funcionales y de información. Se abordan los casos de uso de los distintos actores que intervienen en la aplicación.

El cuarto capítulo aborda la lógica de la aplicación y el desarrollo que se ha llevado a cabo.

2

Arquitectura

2.1. Arquitectura del proyecto

La arquitectura de software define la estructura que un software debe tener, así como las piezas que se deben construir y cómo se deben acoplar y relacionar para trabajar con ellas.
Se centra en el problema que se quiere resolver, estableciendo una jerarquía de clases objetos que se corresponden con las características del mundo real que tienen que ver con el problema.

La arquitectura del software sirve como modelo que proporciona una abstracción describiendo sus componentes y como interactúan entre sí, gestionando su coordinación, comunicación y facilidad o complejidad del propio sistema.
Sin esta arquitectura no sería fácil cumplir con los requisitos que se exponen en el apartado 3, ya que ayuda a poder tomar las decisiones adecuadas para la evolución del proyecto.

Para este proyecto se ha optado por utilizar una arquitectura cliente-servidor.
En la figura uno se observa el diagrama arquitectónico de todos los elementos arquitectónicos que tienen un papel en este proyecto.

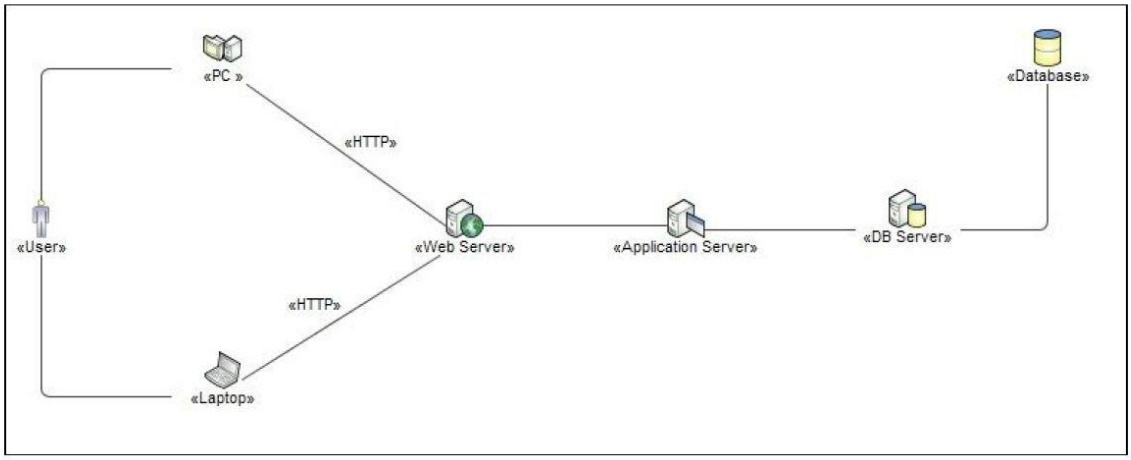


Figura 1: Diagrama arquitectónico de la aplicación

El usuario se puede conectar con un dispositivo que tenga internet habilitado como puede ser un móvil, un pc o un laptop al servidor web. Al acceder a la web, toda la aplicación Angular se instalará en el dispositivo sin necesidad de volver a descargar nada más durante su funcionamiento. Esto es una peculiaridad del framework Angular, que descarga todo el código de la aplicación de una vez para no tener que hacer más peticiones innecesarias. Desde ese momento, se podría decir que el dispositivo actúa de cliente, y a través de esta aplicación descargada se puede conectar al servidor para hacer las peticiones que sean necesarias, según el usuario vaya navegando a través de esta aplicación cliente.

Por cada petición que haga el cliente, se mandarán los datos necesarios al servidor (Application Server, en este caso hecha en NestJS). El servidor verificará todos los datos enviados, y si son correctos procederá a conectarse con el servidor de base de datos, en este caso al entorno de MongoDB (DB Server), el cual se comunica con su propia base de datos (Database). Si los datos enviados no fueran correctos, la aplicación servidor devolverá un mensaje de error al cliente.

A continuación, se muestran una serie de diagramas de secuencia donde se pueden ver distintas peticiones y todo el proceso que se realiza en las mismas.

2.1.2 Diagramas de secuencia

En los siguientes diagramas de secuencia se modela la interacción del usuario y los objetos que intervienen en los distintos procesos de este proyecto.

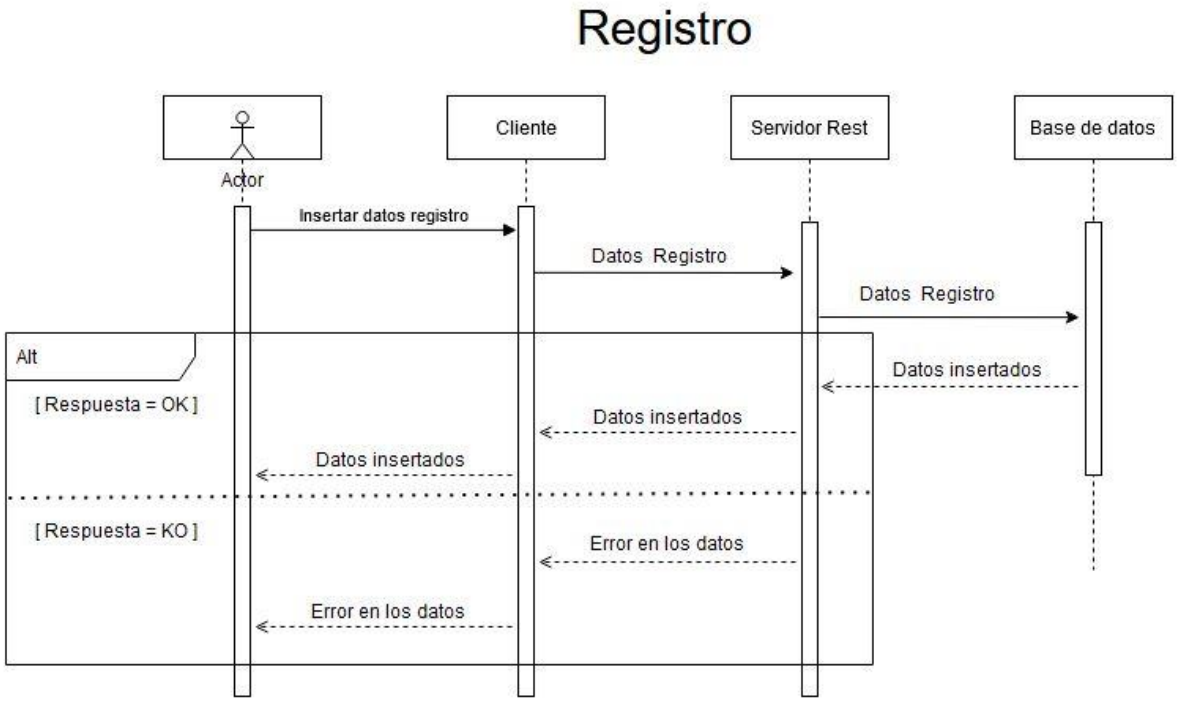


Figura 2: Diagrama de Secuencia del registro

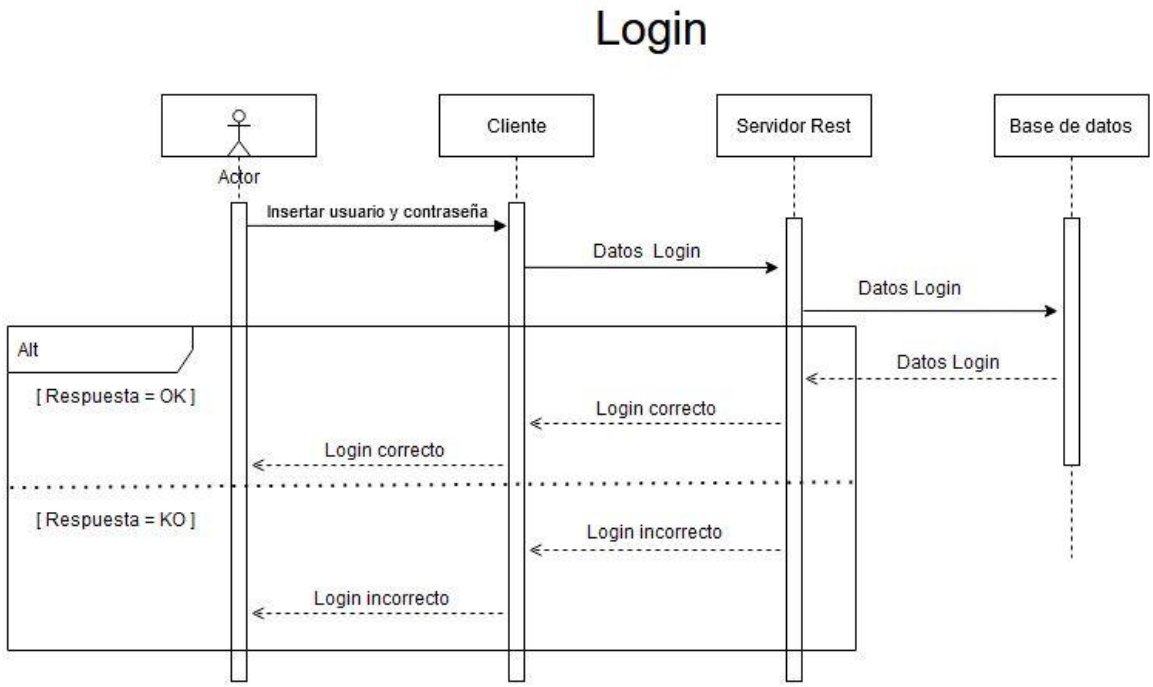


Figura 3: Diagrama de Secuencia del inicio de sesión

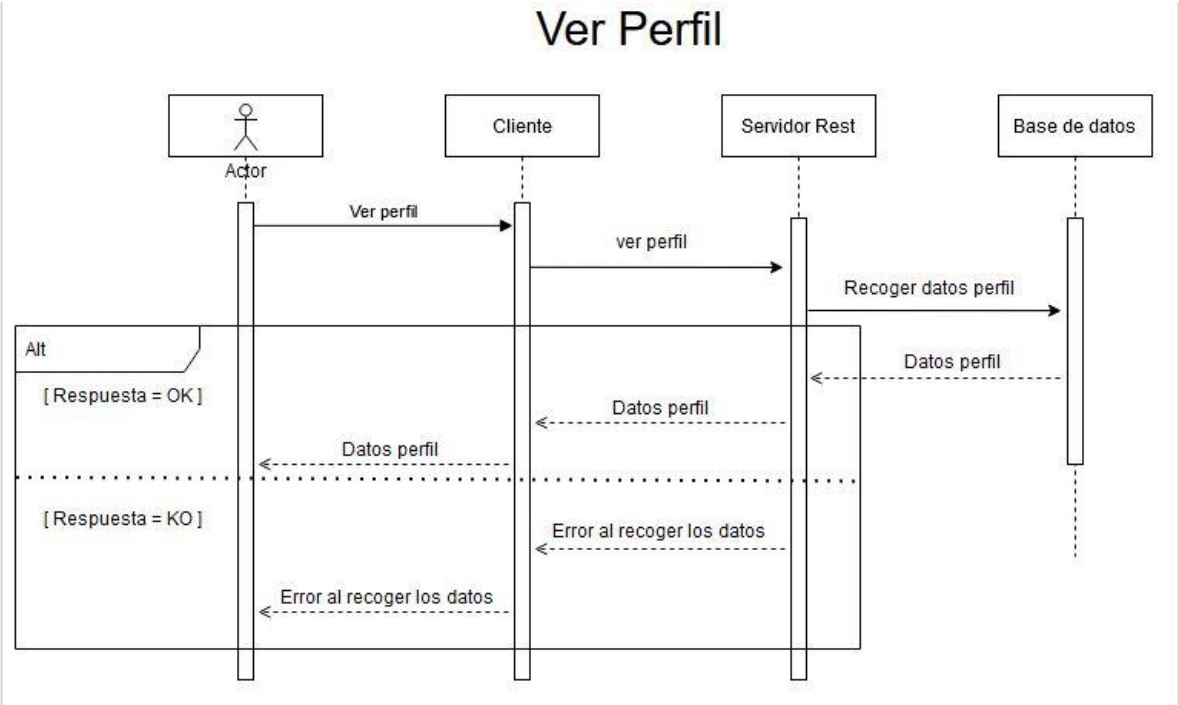


Figura 4: Diagrama de Secuencia para ver el perfil

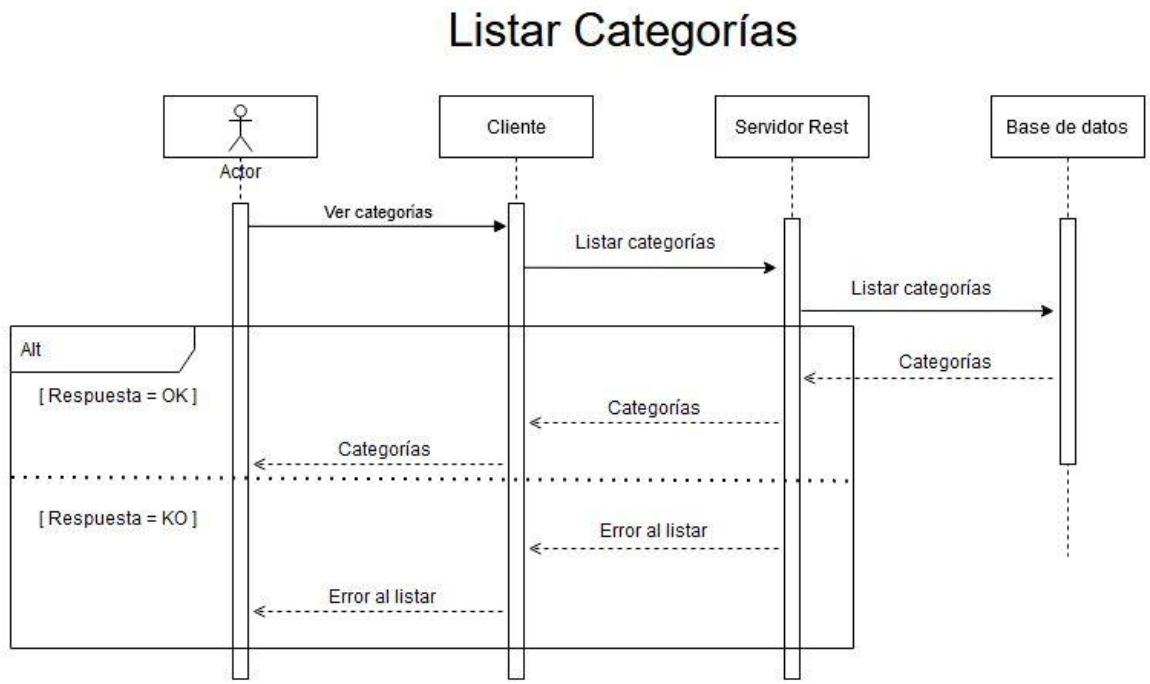


Figura 5: Diagrama de Secuencia al listar categorías

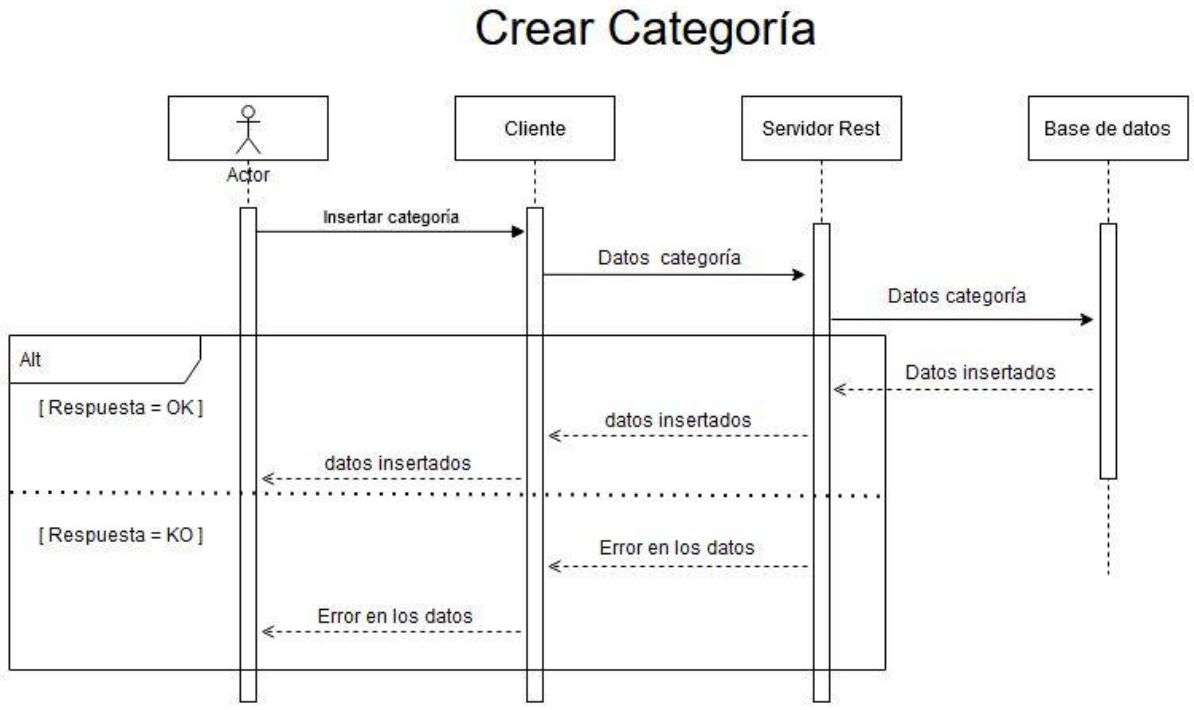


Figura 6: Diagrama de Secuencia para crear categoría

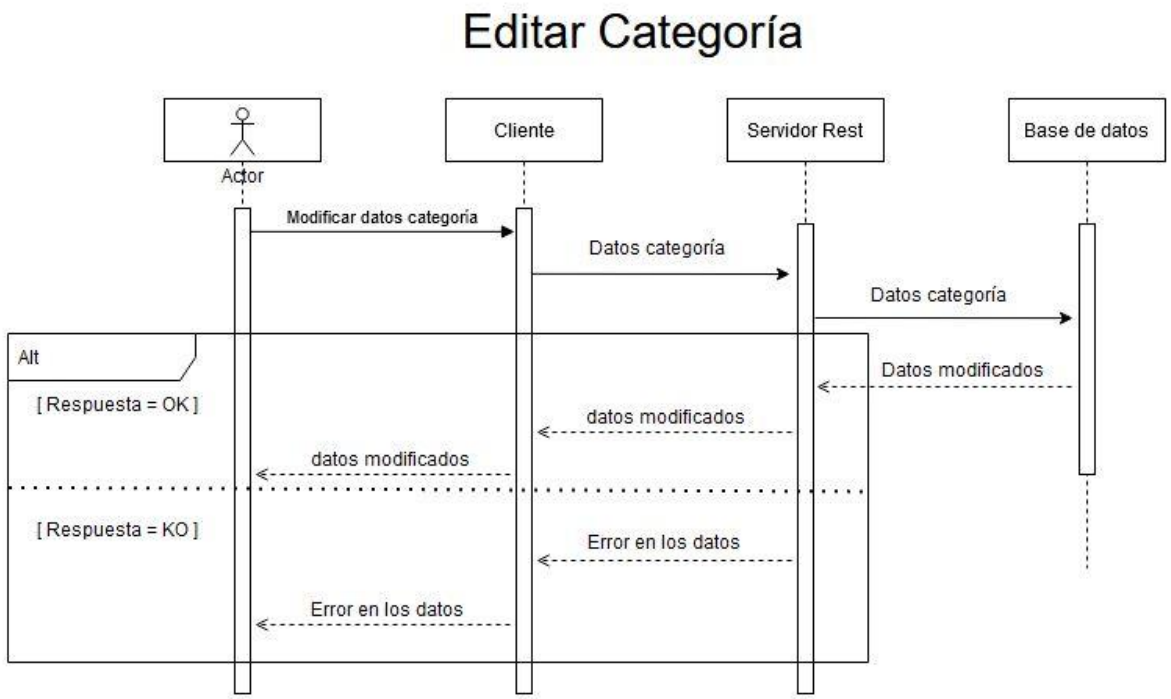


Figura 7: Diagrama de Secuencia para editar categoría

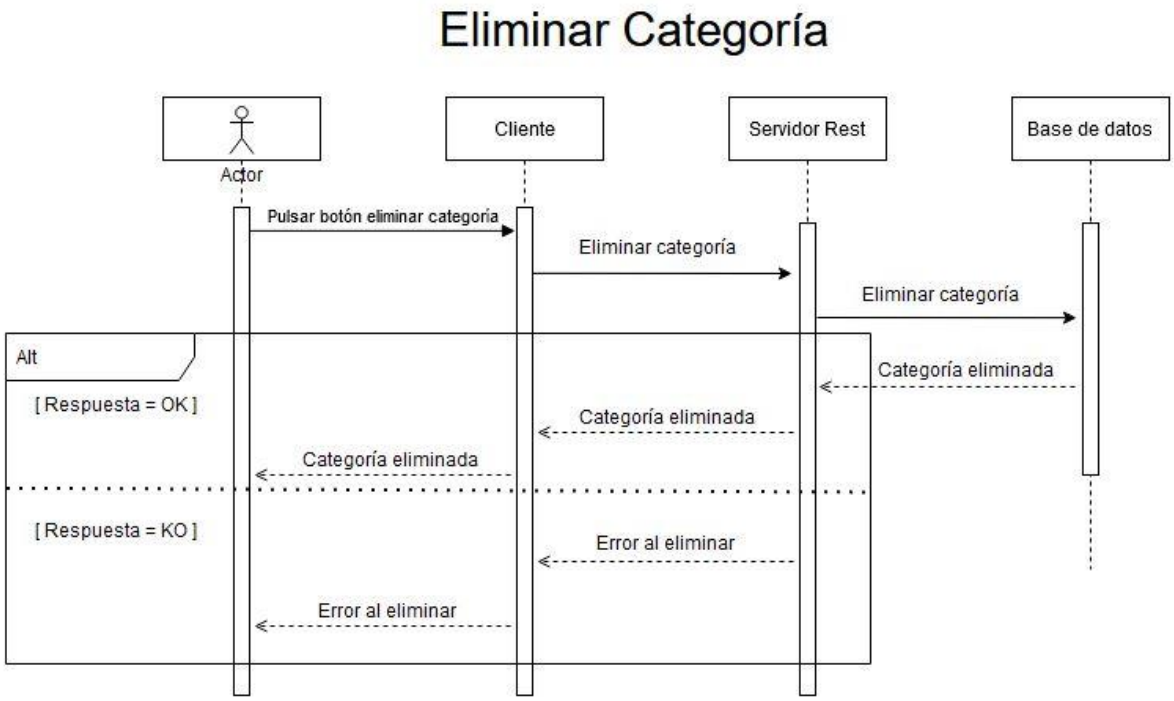


Figura 8: Diagrama de Secuencia al eliminar categoría

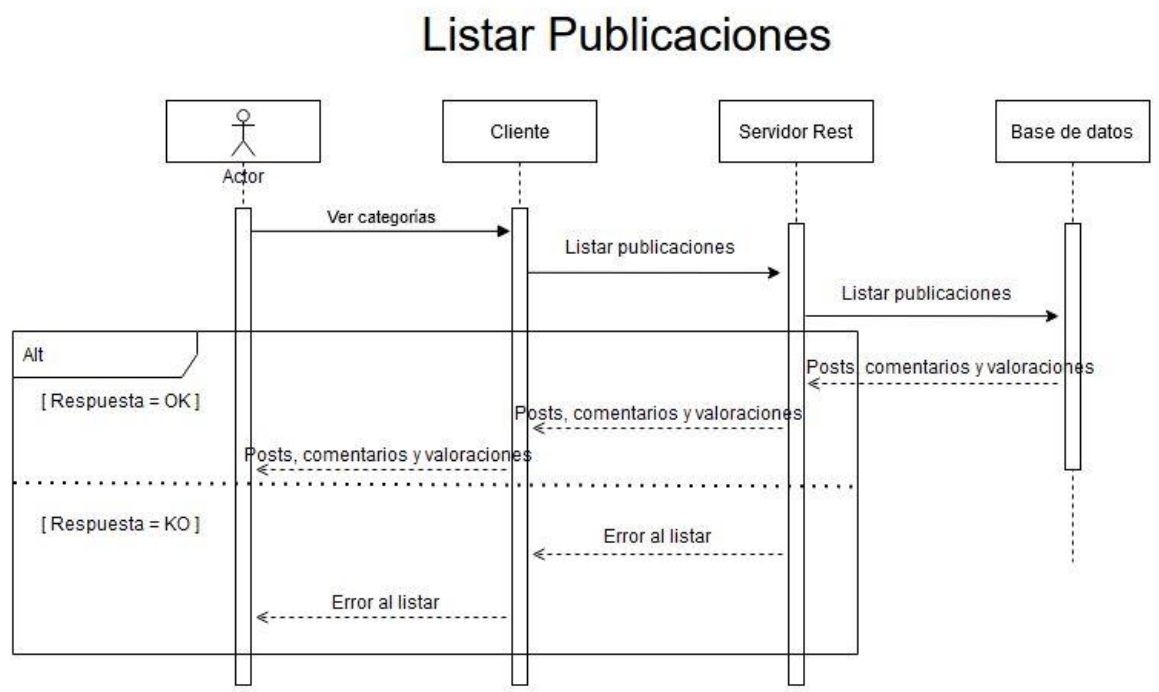


Figura 9: Diagrama de Secuencia al listar las publicaciones

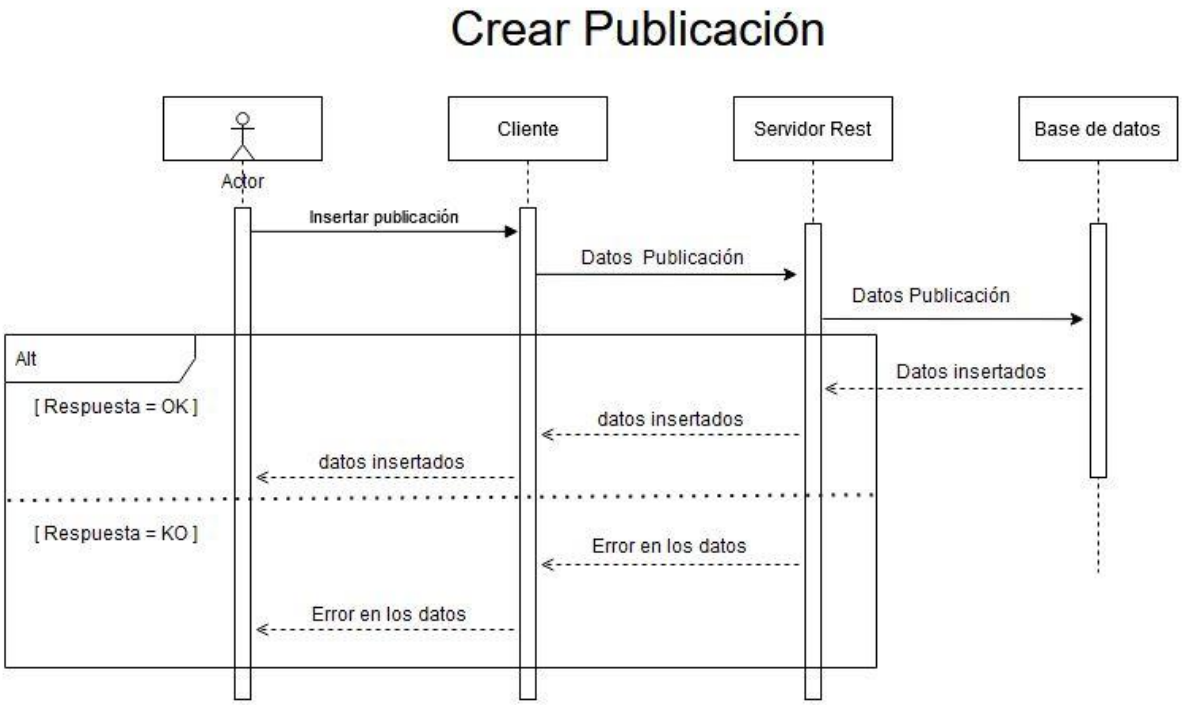


Figura 10: Diagrama de Secuencia al crear publicación

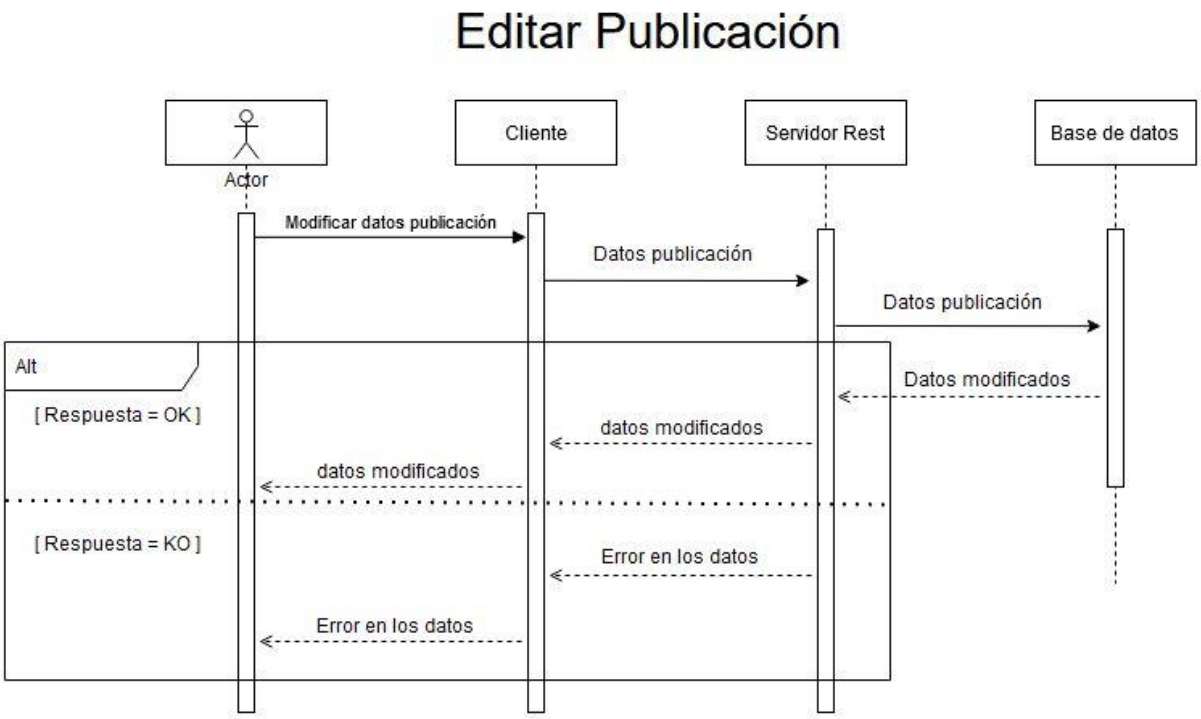


Figura 11: Diagrama de Secuencia al editar una publicación

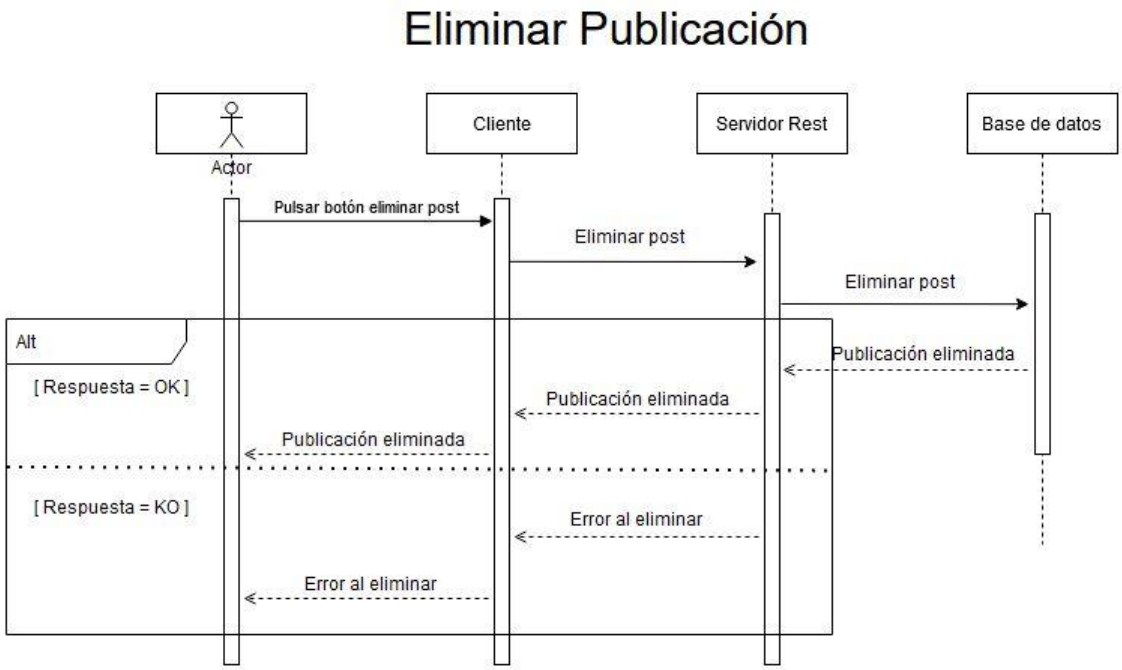


Figura 12: Diagrama de Secuencia al eliminar una publicación

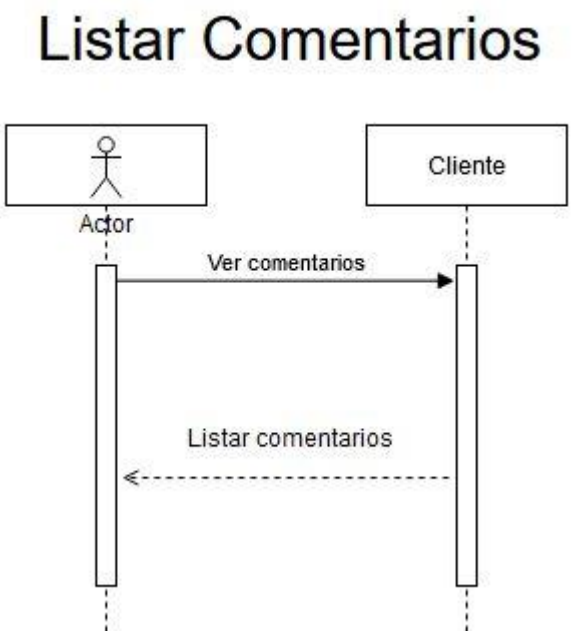


Figura 13: Diagrama de Secuencia al listar comentarios

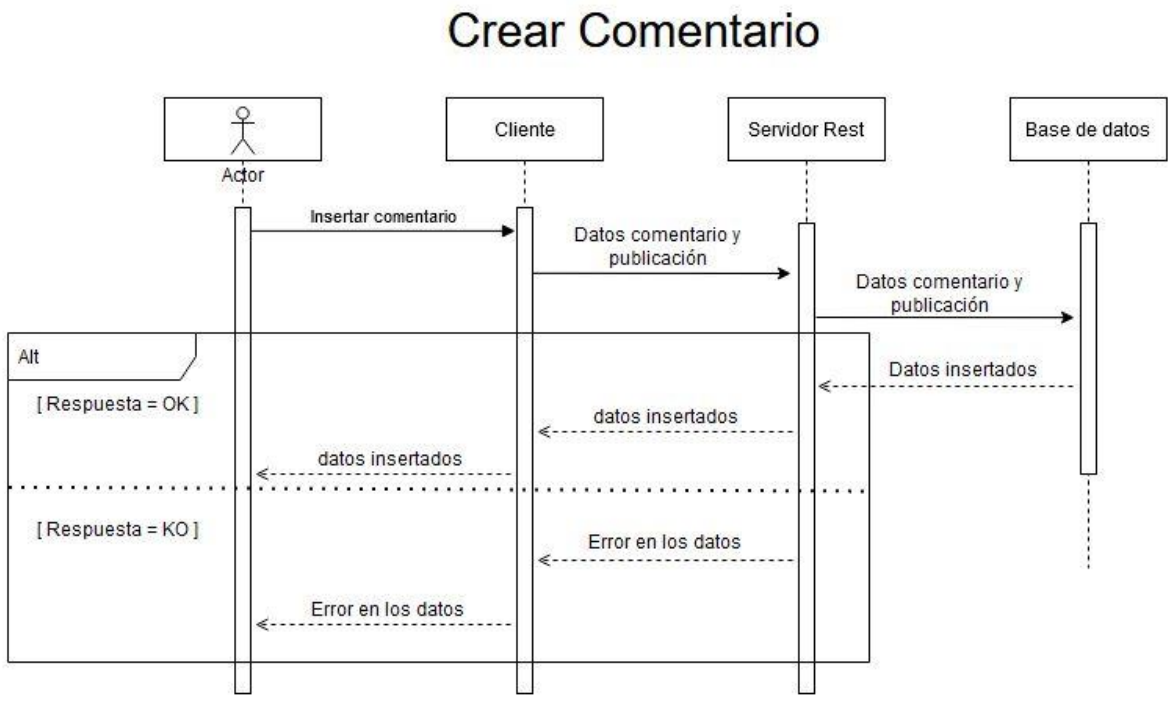


Figura 14: Diagrama de Secuencia para crear un comentario

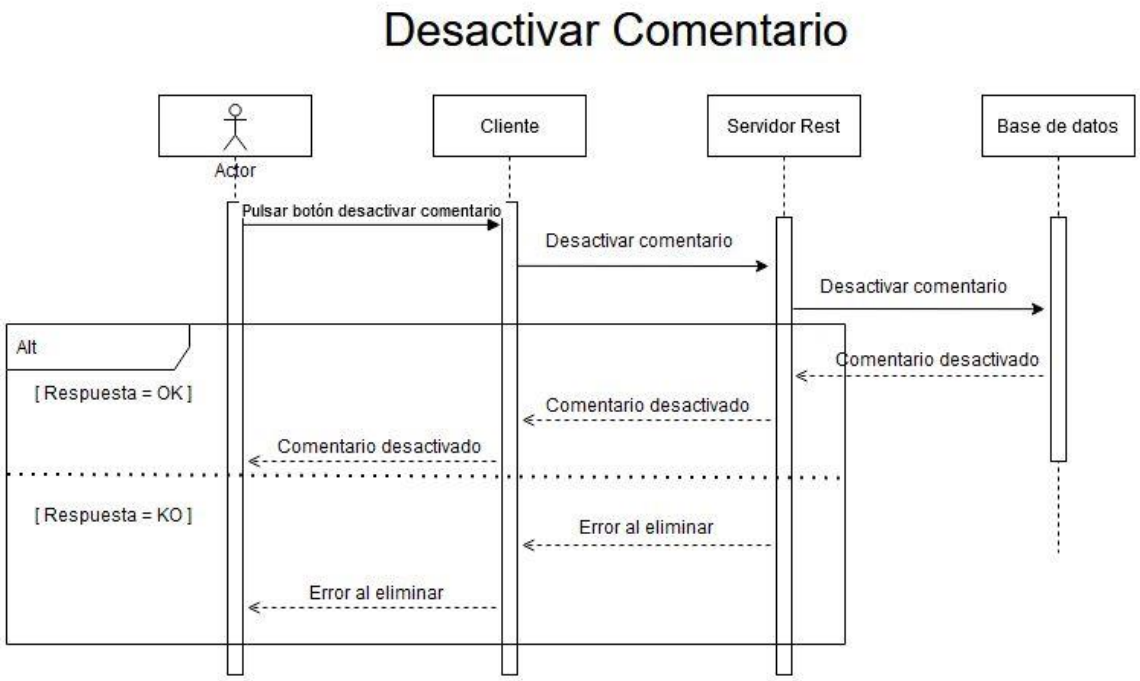


Figura 15: Diagrama de Secuencia al desactivar un comentario

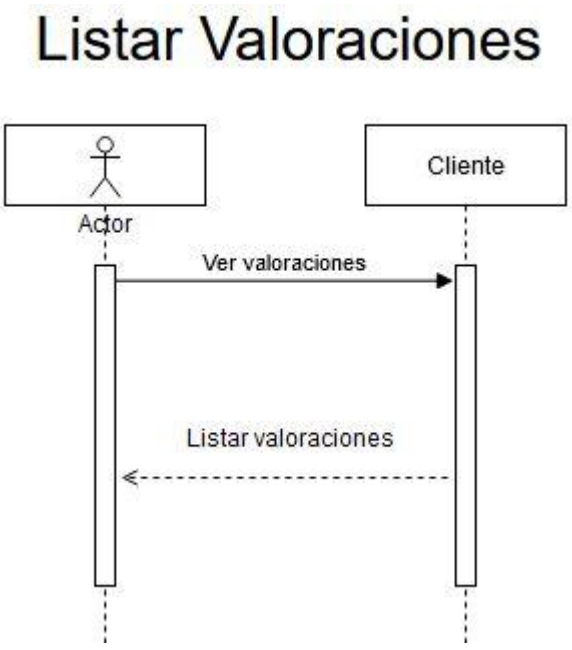


Figura 16: Diagrama de Secuencia al listar las valoraciones de una publicación

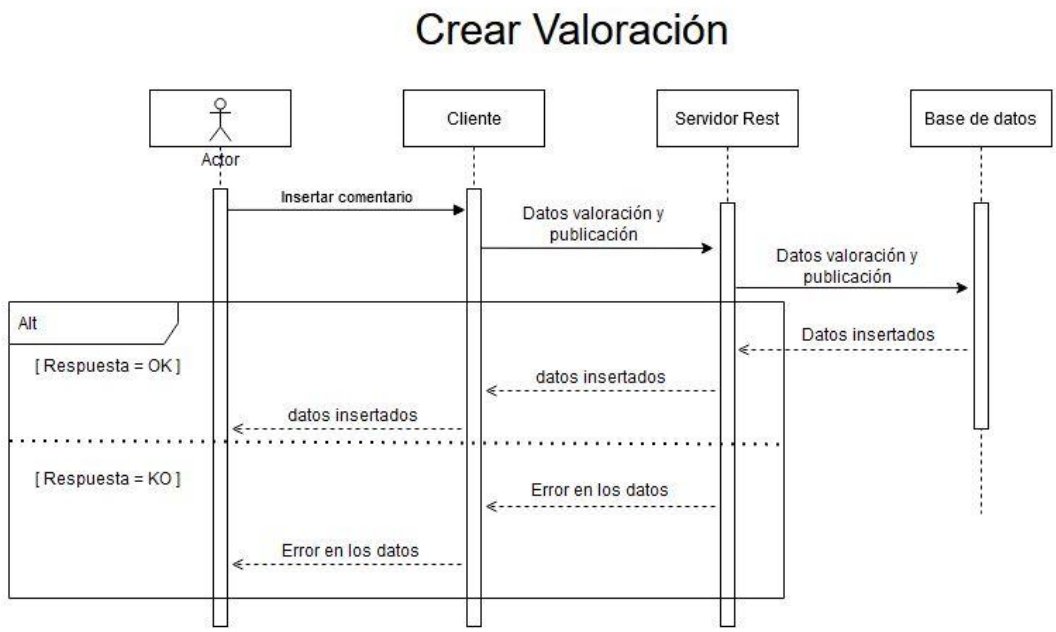


Figura 17: Diagrama de Secuencia cuando se crea una publicación

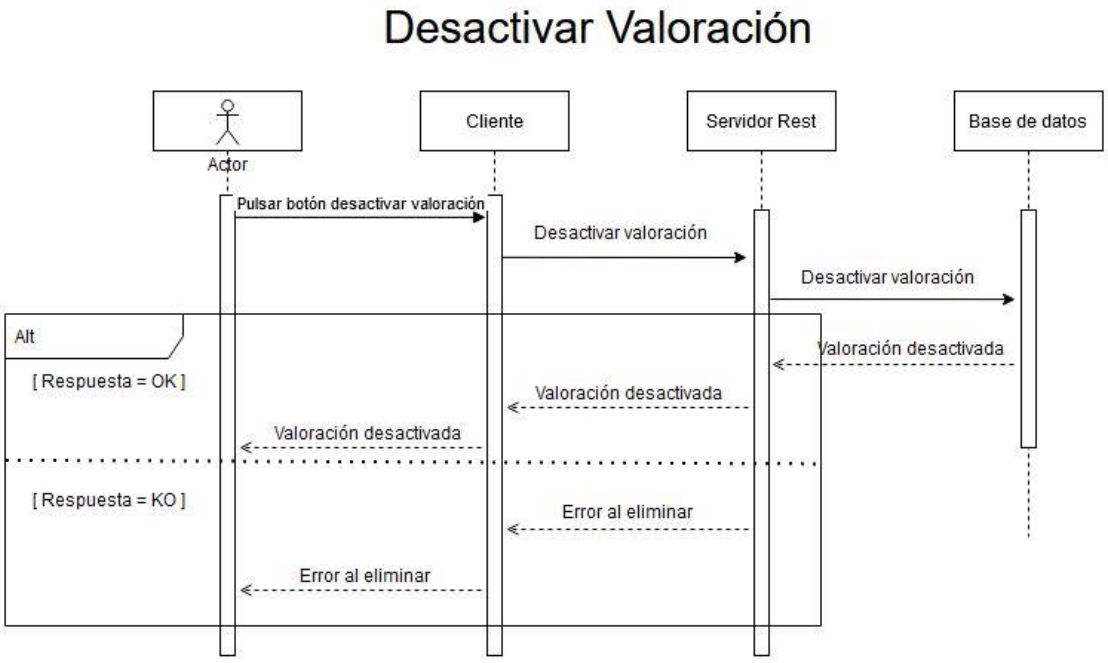


Figura 18: Diagrama de Secuencia al desactivar la valoración de una publicación

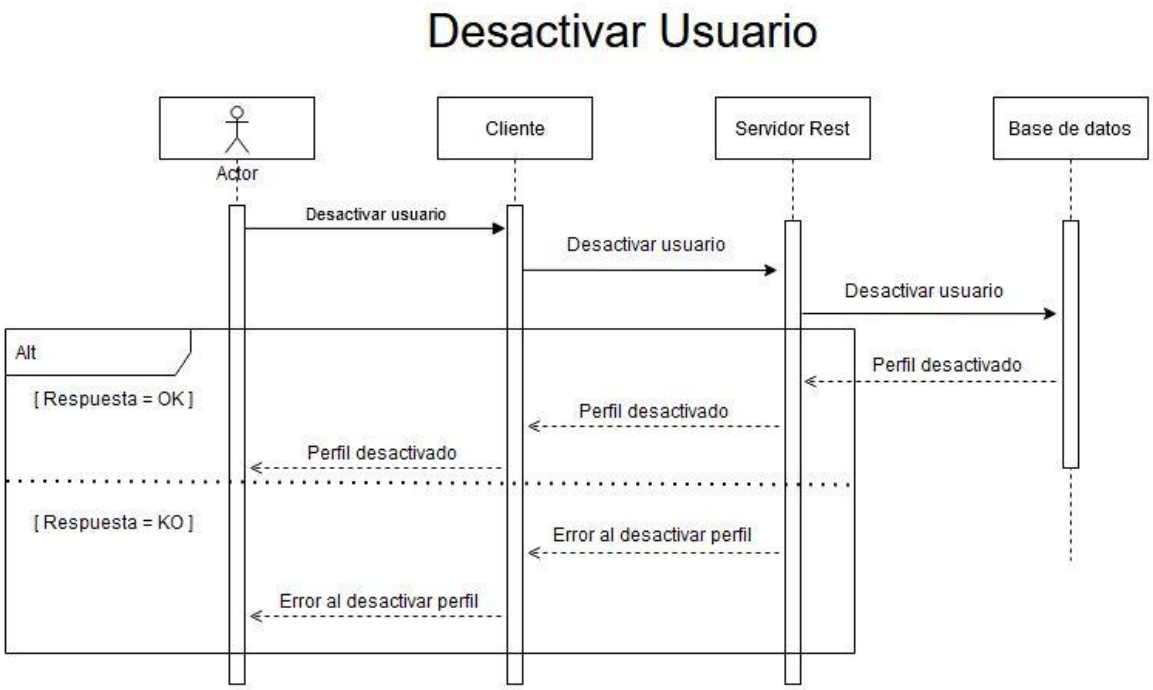


Figura 19: Diagrama de Secuencia al desactivar un usuario

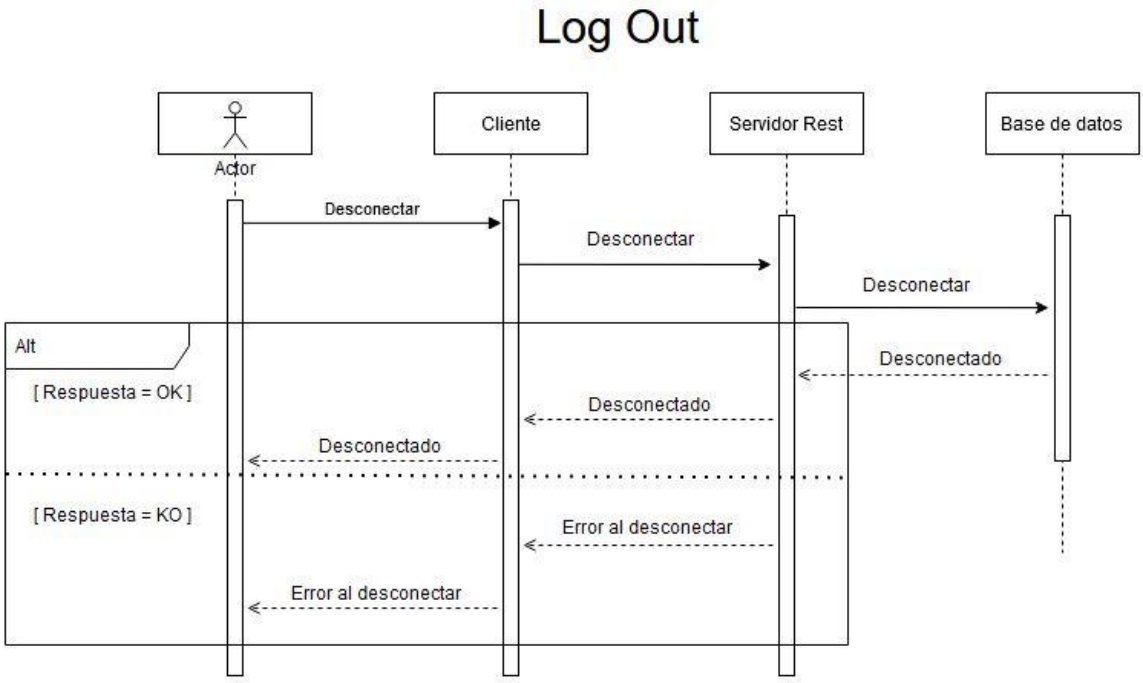


Figura 20: Diagrama de Secuencia al cerrar sesión

2.2. Arquitectura de los entornos de trabajo (frameworks)

2.2.1 Arquitectura en NestJS

El framework usado en el servidor (NestJS) tiene una arquitectura sólida que facilita el desarrollo de aplicaciones escalables y mantenibles.

Esta arquitectura está basada en capas que se centran en la distribución de roles, creando una jerarquía entre los mismos, lo que hace que sea una forma muy efectiva de separar las responsabilidades de los miembros que acceden a la aplicación.

Además, el rol asignado al usuario indica la interacción con las distintas capas, afectando a las responsabilidades que pueden tomar las mismas.

Esta interacción a través de capas permite poder separarlas entre distintas máquinas, pudiendo comunicarse entre componentes a través de interfaces, aunque en este proyecto el servidor será instalado en una sola máquina.

Las capas abstraen la vista del modelo como un todo, teniendo las funcionalidades altamente definidas. Las capas superiores envían órdenes a las capas inferiores, y éstas pueden enviar información a las capas superiores o a otras capas inferiores, además de no tener ningún tipo de dependencia con las superiores, por lo que se pueden reutilizar para otros propósitos y están totalmente desacopladas.

2.2.2 Arquitectura en Angular

Angular usa una arquitectura modelo-vista-modelo de vista (MVVM), un patrón caracterizado por su buen desacoplamiento entre la interfaz del usuario y la lógica de la aplicación.

El modelo representa la capa de datos y lógica de negocio y contiene la información.

La vista representa la información a través de elementos visuales que la componen.

El modelo de vista es una capa intermediaria entre el modelo y la vista. Contiene toda la lógica de presentación y permite la comunicación entre el modelo y la vista.

La principal diferencia entre este modelo y el clásico modelo-vista-controlador radica en que la vista se actualiza automáticamente sin tener que llamar al controlador, además, el modelo de vista permite realizar test de pruebas sin ninguna configuración, por lo que permite mejor testabilidad, reusabilidad y los cambios afectan menos a las vistas.

2.2.3 Arquitectura en MongoDB

MongoDB también usa arquitectura en capas, componiéndose de tres principales:

- Una capa de consulta, que gestiona todas las operaciones *CRUD* (Create, Read, Update, Delete) para insertar, actualizar, eliminar y visualizar datos y que traduce las peticiones realizadas desde el driver de la aplicación cliente (en este caso la aplicación cliente de MongoDB se trata de la aplicación servidor de este proyecto).
- Una capa encargada de gestionar las instrucciones recibidas por la capa de consulta y verificar que las estructuras de datos son correctas. Esta capa también se encarga del mecanismo de replicado, en general es la capa que engloba el modelo de datos.
- Una última capa que gestiona la persistencia de los datos en disco (capa de almacenamiento), compresión, llamadas al sistema etc.

Además de estas 3 capas principales, *MongoDB* tiene otras que se ocupan de la seguridad y de la administración.

El usuario accede a través de un navegador web a la aplicación. En el entorno de producción accedería a la URL donde estuviera alojado el cliente web, pero en este caso se accederá en modo local a la URL localhost:3000. El número 3000 es el puerto local donde la aplicación se ha desplegado.

3

Análisis

Este capítulo expone los requisitos necesarios para implementar esta aplicación, analizando en detalle lo que necesita el cliente y que funcionalidad es necesaria si se quieren lograr las finalidades requeridas.

Se identifican los actores primordiales que van a disponer de la aplicación, a los que se le asigna un rol específico que difiere en una serie de privilegios o permisos sobre las distintas funcionalidades que ofrece la aplicación, exponiendo los roles de forma creciente, donde cada actor tiene los mismos privilegios que el anterior más los específicos de su perfil.

Tras ver los actores se muestran los requisitos funcionales y no funcionales del sistema.

3.1. Catálogo de actores

En la aplicación existen tres tipos de usuarios:

- **Usuario no registrado.** Aquí se engloban a todos los usuarios que no han registrado en la aplicación o no han iniciado sesión en la misma. Puede visualizar el contenido informativo de la aplicación sin hacer uso de las funcionalidades de guardado de datos, modificaciones o borrados.
- **Usuario registrado.** Es el perfil más bajo de usuarios que pueden añadir datos en la aplicación. Tienen la misma funcionalidad que el usuario no registrado a la hora de ver los datos, exceptuando su propia lista de lugares favoritos, con el añadido de poder añadir contenido nuevo. Pueden unificar contenido en distintas listas para tener un rápido acceso al mismo.

- **Administrador.** Es el rol con los permisos más altos. Tiene acceso a todas las piezas de la aplicación, pudiendo editar, crear o borrar cualquier contenido. Dispone de acceso para modificar el rol de otros usuarios.

3.2. Catálogo de requisitos

Se clasifican los requisitos en tres categorías, funcionales, no funcionales y de información.

Los requisitos funcionales de un sistema especifican los servicios, actividades y funcionalidades que son requeridos en dicho sistema. Los requisitos no funcionales imponen restricciones sobre cómo hará esas funciones. Los requisitos de información aclaran los datos necesarios que se usarán en la aplicación.

3.2.1 Requisitos funcionales

3.2.1.1 RF01 Gestión de usuarios

- **RF-01.01 Registro de usuario.** Cualquier usuario se podrá registrar en la aplicación ingresando una serie de datos personales que incluyen un correo electrónico.
- **RF-01-02 Consulta de usuarios.** El usuario administrador podrá ver un listado de los usuarios del sistema.
- **RF-01-03 Editar usuario.** El usuario administrador podrá alterar los datos y el rol del resto de usuarios.
- **RF-01-04 Bloquear usuario.** El usuario administrador podrá bloquear a un usuario impidiendo así su entrada en el sistema.
- **RF-01-05 Inicio sesión de usuario.** Cualquier usuario registrado podrá acceder al inicio de sesión y hacer uso de éste a través de su usuario y contraseña.
- **RF-01-06 Cerrar sesión de usuario.** Cualquier usuario que ha iniciado sesión podrá desconectarse de la misma.
- **RF-01-07 Recuperar contraseña.** Cualquier usuario registrado podrá pedir que se le reenvíe un enlace al correo para poder recuperar la contraseña y modificar la misma.
- **RF-01-08 Modificar perfil.** Los usuarios estándar podrán editar sus propios datos.

3.2.1.2 RF02 Gestión de publicaciones

- **RF-02-01 Consulta de publicaciones.** Los usuarios podrán ver un listado de todas las publicaciones ordenadas de forma descendente por el campo fecha. En este mismo listado se podrán filtrar las publicaciones por distintos campos.
- **RF-02-02 Crear publicación.** Los usuarios registrados podrán crear una publicación añadiendo un nombre de publicación, una localización y pudiendo rellenar una serie de campos opcionales.
- **RF-02-03 Editar publicación.** El usuario administrador podrá editar información sobre una publicación.
- **RF-02-04 Eliminar publicación.** El usuario administrador podrá eliminar una publicación.
- **RF-02-05 Añadir a favoritas.** Los usuarios registrados podrán añadir una publicación a su listado de publicaciones favoritas y acceder a ellas rápidamente en su sección.
- **RF-02-06 Listado de favoritas.** Los usuarios registrados podrán añadir una publicación a su listado de publicaciones favoritas y acceder a ellas rápidamente en su sección.

3.2.1.3 RF03 Gestión de comentarios

- **RF-03-01 Consulta de comentarios.** Los usuarios podrán ver un listado de todos los comentarios ordenados de forma descendente por el campo fecha dentro de cada publicación.
- **RF-03-02 Crear comentario.** Los usuarios registrados podrán añadir un comentario a una publicación existente.
- **RF-03-03 Editar comentario.** El usuario administrador podrá editar cualquier comentario en la aplicación.
- **RF-03-04 Eliminar comentario.** El usuario administrador podrá eliminar cualquier comentario de una publicación.

3.2.1.4 RF04 Gestión de valoraciones

- **RF-04-01 Consulta de valoraciones.** Los usuarios podrán ver en la publicación cuántas valoraciones tiene la misma y la media ponderada que le corresponde.
- **RF-04-02 Emitir valoración.** Los usuarios registrados podrán añadir una valoración a una publicación existente.

3.2.1.5 RF05 Gestión de Categorías

- **RF-05-01 Consulta de categorías.** El usuario administrador tendrá potestad para ver un listado de todas las categorías ordenadas de forma descendente por el campo nombre.
- **RF-05-02 Crear categoría.** El usuario administrador tendrá potestad para añadir una nueva categoría a las ya existentes.
- **RF-05-03 Editar categoría.** El usuario administrador podrá editar el nombre de cualquier categoría en la aplicación.
- **RF-05-04 Eliminar categoría.** El usuario administrador podrá eliminar cualquier categoría en la aplicación.

3.2.1.6 RF06 Gestión de Etiquetas

- **RF-06-01 Consulta de etiquetas.** El usuario administrador podrá consultar si está una etiqueta. Cualquier usuario registrado podrá consultar si está una etiqueta.
- **RF-06-02 Crear etiquetas.** El usuario registrado podrá añadir una nueva etiqueta a las ya existentes.
- **RF-06-03 Editar etiquetas.** El usuario registrado podrá editar el nombre de cualquier etiqueta en la publicación.
- **RF-06-04 Eliminar etiquetas.** El usuario registrado podrá eliminar cualquier etiqueta en la publicación.

3.2.2 Requisitos no funcionales

- **RNF-01 Disponibilidad.** Los servidores de la aplicación deben asegurar una disponibilidad de 24 horas los 7 días de la semana y durante todo el año.
- **RNF-02 Tiempo de respuesta.** La aplicación debe estar preparada para que el tiempo de carga de los datos no supere bajo ningún concepto los 5 segundos.
- **RNF-03 Capacidad de carga.** El sistema debe ser capaz de gestionar al menos 100 usuarios que usen la aplicación simultáneamente.
- **RNF-04 Tolerancia a fallos.** La aplicación debe estar preparada para que no existan errores en la introducción de datos, como pueden ser las fechas.
- **RNF-05 Control de acceso.** Cada usuario no administrador podrá gestionar únicamente las secciones que les corresponden, es decir, sus propios contenidos.
- **RNF-06 Seguridad.** Los usuarios que no son administradores no podrán acceder al panel de administración.
- **RNF-07 Usabilidad.** La aplicación ofrecerá una interfaz gráfica intuitiva propia de una aplicación web, facilitando la navegación y el uso de la misma.
- **RNF-08 Usabilidad II.** La aplicación debe contener un diseño adaptativo a todo tipo de dispositivos tales como ordenadores de escritorio, portátiles, smartphones, televisores, etc.
- **RNF-09 Compatibilidad Software.** La aplicación debe estar preparada para poder ser usada en los navegadores webs más utilizados en la actualidad como son Google Chrome, Mozilla Firefox, Microsoft Edge, Opera, etc.
- **RNF-10 Mantenibilidad.** La aplicación debe estar preparada tras su realización para que el usuario administrador de la misma pueda mantener la web sin necesidad de que el programador tenga que modificar el código, exceptuando que el cliente requiera nuevas funcionalidades.

3.2.2 Requisitos de información

- **RI-01 Usuarios:** Identificador, nombre de usuario, nombre y apellidos, fecha de nacimiento, correo electrónico, fecha de registro, rol, contraseña, activo en la aplicación.
- **RI-02 Categorías:** Identificador, nombre de categoría.
- **RI-03 Etiquetas:** Identificador de publicación, nombre de etiqueta.
- **RI-04 Publicaciones:** Identificador, nombre de la publicación, coordenadas geográficas, descripción, imágenes, creador de la publicación, categoría, fecha de la publicación, etiquetas.
- **RI-05 Comentarios:** Identificador, descripción, publicación asociada, creador del comentario, fecha del comentario.
- **RI-06 Valoraciones:** Identificador, valoración, autor de la valoración.

3.3 Casos de uso

Titulo	**UC-01. Registro de usuario**
Descripción	Autoriza a un usuario registrarse en la aplicación
Pre-condición	No se ha iniciado sesión en la aplicación
Post-condición	Nuevo usuario registrado en la aplicación.
Prioridad	Alta
Actores	Usuario sin registrar.
Escenario principal	
1. El usuario pulsa el botón de registro 2. El sistema muestra una vista con un formulario que contiene los datos del registro. 3. El usuario rellena la información requerida para la creación del nuevo usuario. 4. El usuario pulsa el botón de registro para enviar el formulario. 5. El sistema confirma que el usuario se ha registrado adecuadamente.	

Escenario alternativo
1. El usuario pulsa el botón de registro 2. El sistema muestra una vista con un formulario que contiene los datos del registro. 3. El usuario no rellena la información requerida para la creación del nuevo usuario. 4. El usuario pulsa el botón de registro para enviar el formulario. 5. El sistema muestra un mensaje de error en algunos campos, advirtiendo de que hay algunos que no se han rellenado correctamente.

Figura 21: Caso de uso para el registro de usuario

Titulo	**UC-02. Consulta de usuarios**
Descripción	Permite a un usuario obtener un listado de usuarios.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema muestra un listado de usuarios.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón usuarios. 2. El sistema muestra una vista con el listado de usuarios registrados.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 22: Caso de uso para la consulta de usuarios

Titulo	**UC-03. Editar usuario**
Descripción	Autoriza a un usuario a alterar datos de otro usuario.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema cambia los datos del usuario modificado.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón usuarios. 2. El sistema muestra una vista con el listado de usuarios registrados. 3. El usuario pulsa en el botón "*editar*" de un usuario en concreto. 4. El sistema muestra un formulario con los campos rellenos del usuario a modificar. 5. El usuario modifica los campos que ve pertinentes. 6. El usuario pulsa el botón guardar. 7. El sistema modifica los campos y lanza un aviso de que los campos se han guardado correctamente.	
Escenario alternativo	
1. El usuario pulsa el botón usuarios. 2. El sistema muestra una vista con el listado de usuarios registrados. 3. El usuario pulsa en el botón "*editar*" de un usuario en concreto. 4. El sistema muestra un formulario con los campos rellenos del usuario a modificar. 5. El usuario modifica los campos que ve pertinentes inadecuadamente. 6. El usuario pulsa el botón guardar. 7. El sistema lanza un aviso de error informando de que algún campo no se rellenó correctamente.	

Figura 23: Caso de uso para la edición de un usuario

Titulo	**UC-04. Bloquear usuario**
Descripción	Permite a un usuario bloquear el inicio de sesión de otro usuario.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema bloquea el inicio de sesión de otro usuario.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón usuarios. 2. El sistema muestra una vista con el listado de usuarios registrados. 3. El usuario pulsa en el botón "*editar*" de un usuario en concreto. 4. El sistema muestra un formulario con los campos rellenos del usuario a modificar. 5. El usuario desmarca la casilla de verificación "*activo*" del usuario a modificar. 6. El usuario pulsa el botón guardar. 7. El sistema modifica los campos y lanza un aviso de que los campos se han guardado correctamente.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 24: Caso de uso para la desactivación de un usuario

Titulo	**UC-05. Inicio de sesión**
Descripción	Permite a un usuario iniciar sesión en la aplicación.
Pre-condición	No se ha iniciado sesión en la aplicación y existe un usuario que quiere iniciar sesión con su cuenta.
Post-condición	El usuario inicia sesión en la aplicación.

Prioridad	Alta
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón de iniciar sesión. 2. El sistema muestra una vista con un formulario para el inicio de sesión. 3. El usuario rellena la información requerida para el inicio de sesión. 4. El usuario pulsa el botón de inicio de sesión para enviar el formulario. 5. El sistema confirma que el usuario inició sesión correctamente.	
Escenario alternativo	
1. El usuario pulsa el botón de iniciar sesión. 2. El sistema muestra una vista con un formulario para el inicio de sesión. 3. El usuario no rellena adecuadamente la información requerida para el inicio de sesión. 4. El usuario pulsa el botón de inicio de sesión para enviar el formulario. 5. El sistema muestra un mensaje de error indicando el problema.	

Figura 25: Caso de uso para el inicio de sesión de un usuario

Titulo	**UC-06. Cierre de sesión**
Descripción	Permite a un usuario cerrar sesión en la aplicación.
Pre-condición	Se ha iniciado sesión en la aplicación.
Post-condición	El usuario cierra la sesión en la aplicación.
Prioridad	Alta
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón de cerrar sesión. 2. El sistema cierra la sesión del usuario. 3. El sistema confirma que el usuario cerró sesión.	

Escenario alternativo
No existen escenarios alternativos significativos.

Figura 26: Caso de uso para el cierre de sesión

Titulo	**UC-07. Recuperar contraseña**
Descripción	Permite a un usuario pedir su contraseña.
Pre-condición	No se ha iniciado sesión en la aplicación.
Post-condición	El usuario recupera su contraseña.
Prioridad	Alta
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón de iniciar sesión. 2. El usuario pulsa el botón "*he olvidado mi contraseña*". 3. El sistema muestra un formulario para recuperar la contraseña. 4. El usuario introduce su email en el campo correspondiente. 5. El usuario pulsa el botón "*recuperar contraseña*". 6. El sistema envía un email al correo solicitado con un enlace de recuperación de clave. 7. El usuario pulsa en el enlace del correo o lo copia y lo pega en el navegador. 8. El sistema muestra un formulario para introducir la nueva contraseña. 9. El usuario introduce la nueva contraseña. 10. El sistema lanza un aviso de que la contraseña ha sido modificada.	
Escenario alternativo	
1. El usuario pulsa el botón de iniciar sesión. 2. El usuario pulsa el botón "*he olvidado mi contraseña*". 3. El sistema muestra un formulario para recuperar la contraseña. 4. El usuario introduce su email en el campo correspondiente. 5. El usuario pulsa el botón "*recuperar contraseña*".	

6. El sistema lanza un aviso de que ese correo no se encuentra registrado.

Figura 27: Caso de uso para recuperar la contraseña

Titulo	**UC-08. Modificar perfil**
Descripción	Permite a un usuario modificar sus datos.
Pre-condición	Se ha iniciado sesión en la aplicación.
Post-condición	El sistema cambia los datos del usuario.
Prioridad	Media
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón "*perfil*". 2. El sistema muestra un formulario con los campos rellenos del propio usuario. 3. El usuario modifica los campos que ve pertinentes. 4. El usuario pulsa el botón guardar. 5. El sistema modifica los campos y lanza un aviso de que los campos se han guardado correctamente.	
Escenario alternativo	
1. El usuario pulsa el botón "*perfil*". 2. El sistema muestra un formulario con los campos rellenos del propio usuario. 3. El usuario modifica los campos que ve pertinentes inadecuadamente. 4. El usuario pulsa el botón guardar. 5. El sistema lanza un aviso de que hay algún campo que no se rellenó correctamente.	

Figura 28: Caso de uso para la modificación de los datos del perfil

Titulo	**UC-09. Consulta de publicaciones**
Descripción	Permite a un usuario obtener un listado de publicaciones.
Pre-condición	No son necesarias
Post-condición	El sistema muestra un listado de publicaciones.
Prioridad	Alta
Actores	Usuario no registrado, usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 2.a. El usuario pincha en algún filtro 2.a. El sistema filtra las publicaciones con el filtro elegido	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 29: Caso de uso para consultar las publicaciones

Titulo	**UC-10. Crear publicación**
Descripción	Permite a un usuario crear una nueva publicación.
Pre-condición	Se ha iniciado sesión en la aplicación.
Post-condición	Nuevo publicación creada en la aplicación.
Prioridad	Alta
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón de publicaciones 2. El sistema muestra la vista de la página de publicaciones. 3. El usuario pulsa el botón de nueva publicación.	

4. El sistema muestra un formulario para añadir la nueva publicación. 5. El usuario rellena la información requerida para la creación de la publicación. 6. El usuario pulsa el botón de guardar para enviar el formulario. 7. El sistema confirma que la publicación se ha guardado adecuadamente.
Escenario alternativo
1. El usuario pulsa el botón de publicaciones 2. El sistema muestra la vista de la página de publicaciones. 3. El usuario pulsa el botón de nueva publicación. 4. El sistema muestra un formulario para añadir la nueva publicación. 5. El usuario no rellena la información requerida para la creación de la publicación. 6. El usuario pulsa el botón de guardar para enviar el formulario. 7. El sistema muestra un mensaje de error en algunos campos, advirtiendo de que hay alguno que no se han rellenado correctamente.

Figura 30: Caso de uso para crear una publicación

Titulo	**UC-11. Editar publicación**
Descripción	Permite a un usuario modificar datos de una publicación.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema cambia los datos de la publicación a modificar.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 3. El usuario pulsa en el botón "*editar*" de una publicación concreta. 4. El sistema muestra un formulario con los campos rellenos de la publicación a modificar. 5. El usuario modifica los campos que ve pertinentes.	

6. El usuario pulsa el botón guardar. 7. El sistema modifica los campos y lanza un aviso de que los campos se han guardado correctamente.
Escenario alternativo
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 3. El usuario pulsa en el botón "*editar*" de una publicación concreta. 4. El sistema muestra un formulario con los campos rellenos de la publicación a modificar. 5. El usuario modifica los campos que ve pertinentes inadecuadamente. 6. El usuario pulsa el botón guardar. 7. El sistema lanza un aviso de error informando de que algún campo no se rellenó correctamente.

Figura 31: Caso de uso para editar una publicación

Titulo	**UC-12. Eliminar publicación**
Descripción	Permite a un usuario eliminar una publicación existente.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema elimina la publicación seleccionada.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 3. El usuario pulsa en el botón "*borrar*" de una publicación concreta. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarla. 5. El usuario pulsa el botón "*aceptar*". 7. El sistema elimina la publicación y lanza un aviso de éxito.	

Escenario alternativo
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 3. El usuario pulsa en el botón "*borrar*" de una publicación concreta. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarla. 5. El usuario pulsa el botón "*cancelar*". 7. El sistema no elimina la publicación y vuelve al listado de publicaciones.

Figura 32: Caso de uso para eliminar una publicación

Titulo	**UC-13. Añadir publicación a favoritas**
Descripción	Permite a un usuario añadir una publicación a sus favoritas.
Pre-condición	Se ha iniciado sesión en la aplicación.
Post-condición	La publicación se añade a sus favoritas.
Prioridad	Baja
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 3. El usuario pulsa el botón de favorita en alguna publicación. 4. El sistema añade la publicación al listado de publicaciones favoritas del usuario.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 33: Caso de uso para añadir publicaciones a la lista de favoritos

Titulo	**UC-14. Consulta de publicaciones favoritas**
Descripción	Permite obtener un listado de publicaciones favoritas.
Pre-condición	Se ha iniciado sesión en la aplicación.
Post-condición	El sistema muestra un listado de publicaciones favoritas.
Prioridad	Baja
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 3. El usuario pulsa el botón favoritas. 4. El sistema muestra una vista con las publicaciones favoritas del usuario.	
Escenario alternativo	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes. 3. El usuario pulsa el botón favoritas. 4. El sistema lanza un aviso mostrando que el usuario no tiene publicaciones favoritas.	

Figura 34: Caso de uso para consultar las publicaciones favoritas

Titulo	**UC-15. Consulta de comentarios**
Descripción	Permite a un usuario ver un listado de comentarios.
Pre-condición	Estar en la página de publicaciones
Post-condición	El sistema muestra un listado de comentarios sobre una publicación.
Prioridad	Media
Actores	Usuario no registrado, usuario registrado, administrador.

Escenario principal
1. El usuario pulsa el botón comentarios dentro de una publicación existente. 2. El sistema muestra una vista con el listado de comentarios sobre esa publicación.
Escenario alternativo
1. El usuario pulsa el botón comentarios dentro de una publicación existente. 2. El sistema muestra un aviso advirtiendo de que esa publicación no tiene comentarios.

Figura 35: Caso de uso para la consulta de comentarios en una publicación

Titulo	**UC-16. Crear comentario sobre una publicación**
Descripción	Permite a un usuario crear un comentario en una publicación.
Pre-condición	El usuario debe haber iniciado sesión y estar en la página de publicaciones.
Post-condición	Nuevo comentario creado en la aplicación.
Prioridad	Medio
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón de comentarios en una publicación. 2. El sistema muestra una vista con los comentarios de la publicación y un apartado para rellenar el comentario. 3. El usuario rellena la información requerida para la creación del comentario. 4. El usuario pulsa el botón de enviar para guardar el comentario. 5. El sistema confirma que el comentario se ha guardado adecuadamente.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 36: Caso de uso para generar un comentario acerca de una publicación

Titulo	**UC-17. Editar comentario sobre una publicación**
Descripción	Permite a un usuario modificar datos de un comentario.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema cambia los datos del comentario a modificar.
Prioridad	Media
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón comentarios. 2. El sistema muestra una vista con el listado de comentarios existentes. 3. El usuario pulsa en el botón "*editar*" de un comentario concreto. 4. El sistema muestra un formulario con el comentario relleno a modificar. 5. El usuario modifica el comentario. 6. El usuario pulsa el botón guardar. 7. El sistema modifica el comentario y lanza un aviso de que se ha guardado correctamente.	
Escenario alternativo	
1. El usuario pulsa el botón comentarios. 2. El sistema muestra una vista con el listado de comentarios existentes. 3. El usuario pulsa en el botón "*editar*" de un comentario concreto. 4. El sistema muestra un formulario con el comentario relleno a modificar. 5. El usuario modifica el comentario. 6. El usuario pulsa el botón cancelar. 7. El sistema no modifica el comentario y vuelve al listado de comentarios.	

Figura 37: Caso de uso para editar un comentario

Titulo	**UC-18. Eliminar comentario sobre una publicación**
Descripción	Permite a un usuario eliminar un comentario existente.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema elimina el comentario seleccionado.
Prioridad	Media
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón comentarios. 2. El sistema muestra una vista con el listado de comentarios existentes. 3. El usuario pulsa en el botón "*borrar*" de un comentario concreto. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarlo. 5. El usuario pulsa el botón "*aceptar*". 7. El sistema elimina el comentario y lanza un aviso de éxito.	
Escenario alternativo	
1. El usuario pulsa el botón comentarios. 2. El sistema muestra una vista con el listado de comentarios existentes. 3. El usuario pulsa en el botón "*borrar*" de un comentario concreto. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarlo. 5. El usuario pulsa el botón "*cancelar*". 7. El sistema no elimina el comentario y vuelve al listado de comentarios.	

Figura 38: Caso de uso para eliminar un comentario

Titulo	**UC-19. Consulta de valoración**
Descripción	Permite a un usuario ver la valoración de una publicación.
Pre-condición	No son necesarias
Post-condición	El sistema muestra la valoración de una publicación.

Prioridad	Baja
Actores	Usuario no registrado, usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes que incluye la valoración de cada una de ellas.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 39: Caso de uso para consultar las valoraciones de una publicación

Titulo	**UC-20. Insertar valoración en una publicación**
Descripción	Permite a un usuario añadir una valoración en una publicación.
Pre-condición	Se ha iniciado sesión en la aplicación.
Post-condición	El sistema añade la valoración a una publicación.
Prioridad	Baja
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón publicaciones. 2. El sistema muestra una vista con el listado de publicaciones existentes que incluye la valoración de cada una de ellas. 3. El usuario pulsa sobre una valoración específica para mostrar su satisfacción sobre la publicación. 4. El sistema muestra la valoración añadida por el usuario.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 40: Caso de uso para insertar una valoración acerca de una publicación

Titulo	**UC-21. Consulta de categorías**
Descripción	Permite a un usuario obtener un listado de categorías.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema muestra un listado de categorías.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón categorías. 2. El sistema muestra una vista con el listado de categorías existentes.	
Escenario alternativo	
1. El usuario pulsa el botón categorías. 2. El sistema muestra un aviso de que no hay categorías existentes.	

Figura 41: Caso de uso para la consulta de una categoría

Titulo	**UC-22. Crear categoría**
Descripción	Permite a un usuario crear una nueva categoría.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	Nueva categoría creada en la aplicación.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón de categorías 2. El sistema muestra la vista de la página de categorías. 3. El usuario pulsa el botón de nueva categoría. 4. El sistema muestra un formulario con la información para añadir en la categoría.	

5. El usuario rellena la información requerida para la creación de la categoría. 6. El usuario pulsa el botón de guardar para enviar el formulario. 7. El sistema confirma que la categoría se ha guardado adecuadamente.
Escenario alternativo
No existen escenarios alternativos significativos.

Figura 42: Caso de uso para crear una categoría

Titulo	**UC-23. Editar categoría**
Descripción	Permite a un usuario modificar datos de una categoría.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema cambia los datos de la categoría a modificar.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón categorías. 2. El sistema muestra una vista con el listado de categorías existentes. 3. El usuario pulsa en el botón "*editar*" de una categoría concreta. 4. El sistema muestra un formulario con los campos rellenos de la categoría a modificar. 5. El usuario modifica la categoría. 6. El usuario pulsa el botón guardar. 7. El sistema modifica la categoría y lanza un aviso de que se ha guardado correctamente.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 43: Caso de uso para la edición de una categoría

Titulo	**UC-24. Eliminar categoría**
Descripción	Permite a un usuario eliminar una categoría existente.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	El sistema elimina la categoría seleccionada.
Prioridad	Alta
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón categorías. 2. El sistema muestra una vista con el listado de categorías existentes. 3. El usuario pulsa en el botón "*borrar*" de una categoría concreta. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarla. 5. El usuario pulsa el botón "*aceptar*". 7. El sistema elimina la categoría y lanza un aviso de éxito.	
Escenario alternativo	
1. El usuario pulsa el botón categorías. 2. El sistema muestra una vista con el listado de categorías existentes. 3. El usuario pulsa en el botón "*borrar*" de una categoría concreta. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarla. 5. El usuario pulsa el botón "*cancelar*". 7. El sistema no elimina la categoría y vuelve al listado de categorías.	

Figura 44: Caso de uso para la eliminación de una categoría

Titulo	**UC-25. Consulta de etiquetas**
Descripción	Permite a un usuario consultar una publicación por etiqueta.
Pre-condición	No son necesarias.
Post-condición	El sistema muestra un listado de etiquetas.

Prioridad	Baja
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón etiquetas. 2. El sistema muestra una vista con el listado de etiquetas existentes.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 45: Caso de uso para la consulta de etiquetas

Titulo	**UC-26. Crear etiqueta**
Descripción	Permite a un usuario crear una nueva etiqueta.
Pre-condición	Se ha iniciado sesión en la aplicación y está en la página de publicaciones.
Post-condición	Nueva etiqueta creada en la publicación.
Prioridad	Baja
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón de nueva publicación 2. El sistema muestra la vista de la página de publicaciones. 3. El usuario rellena la información requerida para la creación de la publicación, insertando las etiquetas que quiere que estén contenidas en la publicación. 4. El usuario pulsa el botón de guardar para enviar el formulario. 5. El sistema confirma que la publicación se ha guardado adecuadamente y añade las etiquetas que no formaban parte del sistema de etiquetado.	
Escenario alternativo	
1. El usuario pulsa el botón de publicaciones	

2. El sistema muestra la vista de la página de publicaciones. 3. El usuario no rellena la información requerida para la creación de la publicación. 4. El usuario pulsa el botón de guardar para enviar el formulario. 5. El sistema muestra un mensaje de error en algunos campos, advirtiendo de que hay alguno que no se han rellenado correctamente.

Figura 46: Caso de uso para crear etiquetas

Titulo	**UC-26. Crear etiqueta II**
Descripción	Permite a un usuario crear una nueva etiqueta.
Pre-condición	Se ha iniciado sesión en la aplicación con rol de administrador.
Post-condición	Nueva etiqueta creada en la aplicación.
Prioridad	Baja
Actores	Administrador.
Escenario principal	
1. El usuario pulsa el botón de etiquetas. 2. El sistema muestra la vista de la página de etiquetas ordenadas por nombre. 3. El usuario pulsa el botón de nueva etiqueta. 4. El sistema muestra un formulario con la información para añadir en la etiqueta. 5. El usuario rellena una nueva etiqueta. 6. El usuario pulsa el botón de guardar para enviar el formulario. 7. El sistema confirma que la etiqueta se ha guardado adecuadamente.	
Escenario alternativo	
No existen escenarios alternativos significativos.	

Figura 47: Caso de uso para crear etiquetas

Titulo	**27. Editar etiqueta**
Descripción	Permite a un usuario modificar una etiqueta.
Pre-condición	Se ha iniciado sesión en la aplicación.
Post-condición	El sistema cambia los datos de la etiqueta a modificar.
Prioridad	Baja
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón etiquetas. 2. El sistema muestra una vista con el listado de etiquetas existentes. 3. El usuario pulsa en el botón "*editar*" de una etiqueta concreta. 4. El sistema muestra un formulario para modificar la etiqueta. 5. El usuario modifica la etiqueta. 6. El usuario pulsa el botón guardar. 7. El sistema modifica la etiqueta y lanza un aviso de que se ha guardado correctamente.	
Escenario alternativo	
1. El usuario pulsa el botón etiquetas. 2. El sistema muestra una vista con el listado de etiquetas existentes. 3. El usuario pulsa en el botón "*editar*" de una etiqueta concreta. 4. El sistema muestra un formulario para modificar la etiqueta. 5. El usuario pulsa el botón cancelar. 7. El sistema no guarda la etiqueta y vuelve al listado de etiquetas.	

Figura 48: Caso de uso para editar etiquetas

Titulo	**28. Eliminar etiqueta**
Descripción	Permite a un usuario eliminar una etiqueta existente.
Pre-condición	Se ha iniciado sesión en la aplicación y está editando la publicación.
Post-condición	El sistema elimina la etiqueta seleccionada.
Prioridad	Baja
Actores	Usuario registrado, administrador.
Escenario principal	
1. El usuario pulsa el botón etiquetas. 2. El sistema muestra una vista con el listado de etiquetas existentes. 3. El usuario pulsa en el botón "*borrar*" de una etiqueta concreta. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarla. 5. El usuario pulsa el botón "*aceptar*". 7. El sistema elimina la etiqueta y lanza un aviso de éxito.	
Escenario alternativo	
1. El usuario pulsa el botón etiquetas. 2. El sistema muestra una vista con el listado de etiquetas existentes. 3. El usuario pulsa en el botón "*borrar*" de una etiqueta concreta. 4. El sistema muestra un aviso preguntando si está seguro de querer borrarla. 5. El usuario pulsa el botón "*cancelar*". 7. El sistema no elimina la etiqueta y vuelve al listado de etiquetas.	

Figura 49: Caso de uso para eliminar etiquetas

4

Implementación

En el presente capítulo se hablará de la metodología usada para la implementación de esta aplicación, así como de la estructura de los archivos necesarios para que esté en funcionamiento.

Gracias a este capítulo, si se desea ampliar la funcionalidad de la aplicación es más sencillo conocer dónde están los archivos necesarios para realizar modificaciones, o bien, ampliar la funcionalidad de la propia aplicación.

4.1. Metodología de desarrollo

Para el desarrollo del proyecto se ha utilizado la metodología ágil *SCRUM*.

Scrum es una metodología de trabajo que se utiliza en proyectos de gran complejidad. Gracias a esta metodología, se puede generar un gran resultado de forma rápida y eficiente, por lo que es ideal para proyectos que necesitan de cierta flexibilidad.

Aunque es una metodología que se está aplicando mucho a los equipos de trabajo tecnológicos, Scrum permite aplicar su método a cualquier proyecto que se trabaje en equipo.

SCRUM aprovecha la flexibilidad, productividad, tiempo y colaboración del equipo, con el objetivo de conseguir los mejores resultados en el menor tiempo posible. Está caracterizado por el acogimiento de una táctica de desarrollo incremental y la superposición de los distintos ciclos del desarrollo, en lugar de ejecutar cada una de estas fases una tras otra en un ciclo secuencial o en cascada.

El *sprint* es el núcleo central de esta metodología y se trata de cada una de las fases o iteraciones que vamos a tener el el proyecto, consiguiendo un incremento de valor en el mismo.

Para este proyecto se ha dividido el tiempo en 5 *sprint*, fases, iteraciones o ciclos.
Cada uno de estos sprint, desarrollará una serie de historias de usuario que afectan a un conjunto de requisitos que posibilitan desarrollar la funcionalidad requerida.

Uno de los puntos fuertes de *SCRUM* es la asignación de roles para el trabajo. Están divididos en 3 categorías principales:

- El *Product Owner* es el responsable de estar en contacto directo con el cliente. Es necesario que se reúna periódicamente con el equipo de trabajo para verificar que el proyecto avanza adecuadamente. El tutor del proyecto es quien asume el papel de *Product Owner*.

- Scrum Master o facilitador de proyectos se encarga de liderar los equipos, proporcionando la gestión necesaria para todo el proyecto. Es fundamental que este rol minimice todas las dificultades que el equipo de desarrollo pueda encontrarse para poder alcanzar los objetivos necesarios en el tiempo programado.

- El equipo de desarrollo con todos los miembros del proyecto que trabajan conjuntamente para aumentar el producto en cada *sprint*. En este grupo estarían los desarrolladores, testers y demás miembros del equipo de trabajo.

Este proyecto se ha dividido en 6 *sprints*, con una duración de 50 horas cada uno. Al final de cada uno de los *sprints* se realiza una reunión con el *Product Owner* para verificar que el proyecto sigue las pautas adecuadamente.

A su vez, todos los *sprints* se subdividen en 5 fases:

- Análisis de las necesidades y características asociadas a cada funcionalidad de la aplicación que requiere el cliente, así como su especificación.
- Diseño con una descripción de la estructura interna del software.
- Implementación de la programación de cada una de las tareas especificadas centrándose en el diseño previo.
- Cotejamiento y legitimación de la funcionalidad del sistema.
- Documentación actualizada progresiva de este documento.

Las iteraciones 1, 2 y 3 formarán parte del desarrollo de la parte del servidor. En ellas se ha creado el repositorio para esta parte de la aplicación, y se ha ido desarrollando la API así como la base de datos en la nube.

En las iteraciones 4, 5 y 6 se ha desarrollado la parte del cliente que realiza las peticiones a la API, donde el usuario final puede ver la aplicación en funcionamiento.

4.2. Implementación Servidor

4.2.1. Base de datos MongoDB

Para la creación de la base de datos se ha optado por una base de datos no relacional. Para ello se ha creado un clúster en https://cloud.mongodb.com. A través de la aplicación server se han generado las colecciones necesarias para poder guardar todos los datos en esta base de datos.

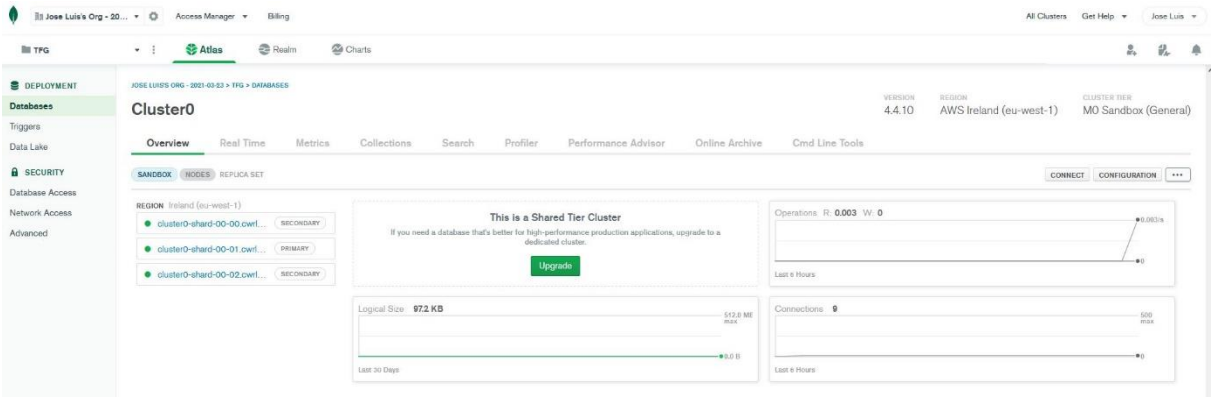


Figura 50: Cluster MongoDB

Para la conexión de esta base de datos no relacional con nuestro proyecto servidor, será necesario instalar *Mongoose*, un paquete disponible en *NPM* con el que podremos hacer las conexiones a la base de datos de una forma simple.

4.2.2. API

Para la creación de la API se ha usado el framework NEST. A través de la consola de comandos (cli) se han ejecutado dos comandos para iniciar la instalación.

$npm i -g @nestjs/cli es un comando que se usa para instalar las dependencias del framework y poder usar todos los comandos que éste ofrece.

$nest new server-tfg se usa para crear un nuevo proyecto con este framework, este comando genera todos los archivos básicos para poder empezar a trabajar.

Al haber instalado las dependencias, *Nest* nos ofrece una serie de comandos propios con los que podemos generar distintos archivos y obtener ciertas funcionalidades que harán falta para poder crear la API.

Usando el comando *npm start* se arranca la aplicación, por defecto lo hace en el puerto 3000, aunque se puede cambiar si se desea otro.

Entrando a localhost:3000 se puede ver como la aplicación ya está en marcha y tiene creada una pequeña página de prueba para ver que todo está funcionando correctamente.

Los principales comandos que se usan para generar los archivos son los siguientes:

- *nest g mo nombre* para generar el módulo que se deseé. Creará una carpeta con el nombre asignado y un archivo que contendrá el módulo generado.
- *nest g co nombre* generará el archivo controlador. Al ponerle el mismo nombre que el módulo lo meterá en la misma carpeta.
- *nest g s nombre* generará el archivo servicio. Al ponerle el mismo nombre que el módulo lo meterá en la misma carpeta.
- *nest g class path/paht/nombre* clase generará una clase de *typescript* con la nomenclatura deseada en la ruta "*path*".

Más adelante se explica para qué sirven todos estos archivos y se mostrarán partes del código para entender la funcionalidad de cada uno de ellos.

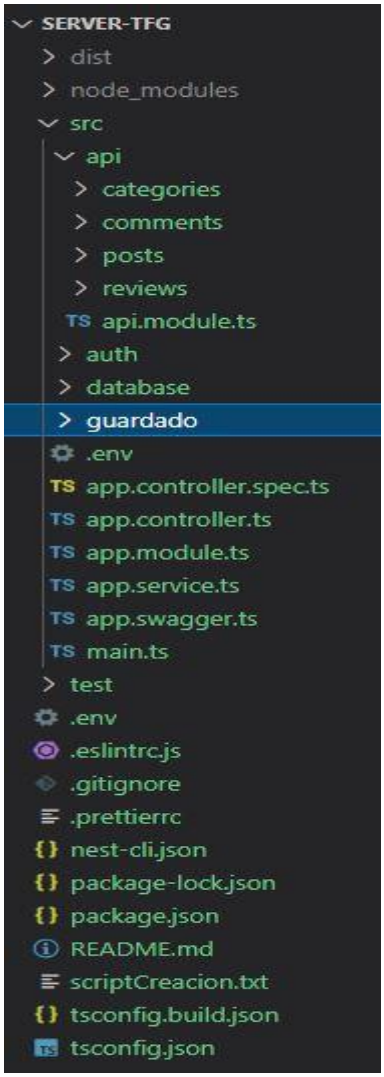


Figura 51: Estructura directorios servidor

En la figura 51 se observan todos los archivos existentes para que funcione la aplicación server.

Dentro de la carpeta api se han creado carpetas perfectamente diferenciadas para saber de qué funcionalidad se ocupa cada una. Cabe destacar que los archivos .spec.ts son archivos de pruebas y aunque no son necesarios para la funcionalidad de la aplicación, si nos ayudan para poder identificar errores y hacer pruebas.

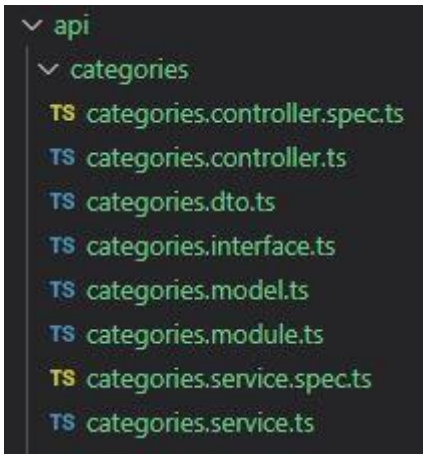


Figura 52: Directorio categorías servidor

Dentro de la carpeta que contiene la funcionalidad deseada (en este caso categories se ocupa de la funcionalidad de las categorías) tenemos varios archivos.

En cada carpeta de este tipo, existe un archivo llamado controller.ts. Este archivo es el controllador de esta parte de la aplicación, y se encarga de poner en contacto los archivos del modelo con la vista. Define claramente la ruta a utilizar como punto de entrada para cada funcionalidad deseada, hace una petición al modelo y devuelve los datos de la vista, en este caso en formato JSON ya que es una API.

Cada controlador tiene un método constructor que genera una instancia de un objeto de este tipo. También tiene distintos métodos que pueden ser de tipo GET, POST, PUT o DELETE.

```
@Get()
async getCategories(): Promise<CategoryDTO[]>  {
    return await this.categoryService.getCategories();
}

@Get(':id')
async getCategory(@Param('id') id): Promise<CategoryDTO> {
    const cat = await this.categoryService.getCategory(id);
    if(!cat)
        throw new NotFoundException("Category not found");
    return cat;
}
```

Figura 23: Métodos Get

Las funciones con método Get sólo devuelven datos que han sido introducidos previamente en la base de datos. Podemos pedir que se devuelvan todos los registros de una colección, o bien uno en particular. Estos métodos no suelen tener ningún tipo de protección al llamarlos, ya que no es necesario que el usuario esté registrado para poder ver los datos que ya hay introducidos en las colecciones, a excepción de algunos en concreto, como pueden ser los registros de usuarios.

```
@UseGuards(AuthGuard('jwt'))
@ApiBearerAuth()
@Post()
async createCategory(@User() user: any, @Res() res: any, @Body() categoria:CategoryDTO ): Promise<CategoryDTO>{
    const actualUser = await this.userService.findUserByUsername(user.username);
    if(!actualUser.admin || !actualUser.active){
        return res.status(HttpStatus.UNAUTHORIZED).json({
            status: 401,
            message: 'Unauthorized'
        });
    }

    const cat = await this.categoryService.createCategory(categoria);
    return res.status(HttpStatus.OK).json({
        message: 'Category created',
        category: cat
    });
}
```

Figura 54: Métodos Post

Los métodos post son los encargados de ponerse en contacto con el modelo para insertar información en la base de datos. Para ello, es necesario que el usuario que va a usarlo esté previamente dado de alta, y tenga los permisos adecuados para poder insertar esa información. Como método de protección en este tipo de métodos (no sólo en POST, sino también en DELETE y PUT) se genera un token único que el usuario debe mandar al hacer la petición. Este token se genera en el inicio de sesión del usuario y guarda en su ordenador para que pueda usarlo en los métodos necesarios.

```
@UseGuards(AuthGuard('jwt'))
@ApiBearerAuth()
@Put(':id')
async updateCategory(@User() user: any, @Res() res: any, @Body()  categoria:CategoryDTO, @Param('id') id): Promise<CategoryDTO>{
    const actualUser = await this.userService.findUserByUsername(user.username);

    if(!actualUser.admin || !actualUser.active){
        return res.status(HttpStatus.UNAUTHORIZED).json({
            status: 401,
            message: 'Unauthorized'
        });
    }

    const cat = await this.categoryService.updateCategory(id, categoria);
    return res.status(HttpStatus.OK).json({
        message: 'Category updated',
        category: cat
    });
}
```

Figura 55: Método Put

Las funciones con método PUT se encargan de modificar registros previamente añadidos a una colección.

```
@UseGuards(AuthGuard('jwt'))
@ApiBearerAuth()
@Delete(':id')
async deleteCategory(@User() user: any, @Res() res: any, @Param('id') id): Promise<CategoryDTO>{
    const actualUser = await this.userService.findUserByUsername(user.username);
    if(!actualUser.admin || !actualUser.active){
        return res.status(HttpStatus.UNAUTHORIZED).json({
            status: 401,
            message: 'Unauthorized'
        });
    }

    const cat = await this.categoryService.deleteCategory(id);
    return res.status(HttpStatus.OK).json({
        message: 'Category deleted',
        category: cat
    });
}
```

Figura 56: Método Delete

Otra función indispensable es la que usa el método delete, encargado de poder pedir al modelo que elimine los datos que se le envían. Al igual que los métodos de inserción y edición, el de borrado requiere de un token previo y una autorización de nivel para poder ser usado.

```
import { Schema } from "mongoose";

export const CategorySchema = new Schema(
    {
        name: { type: String, required: true }
    }
);
```

Figura 57: Clase Model

```
import { Document } from "mongoose";

export interface ICategory extends Document{
    readonly name: string;
}
```

Figura 58: Interfaz Modelo

```
import { IsString, MinLength, MaxLength } from "class-validator";

export class CategoryDTO{
    @IsString()
    @MinLength(3)
    @MaxLength(50)
    readonly name: string;
}
```

Figura 59: Clase DTO

El modelo de cada funcionalidad dentro de esta carpeta está formado por varios archivos (.dto, .interface, .model). Estos archivos se encargan tanto de crear los campos necesarios en la base de datos MongoDB, como de hacer las peticiones a la misma para que ejecute las sentencias que hagan falta. Hace uso de distintas restricciones y validaciones para verificar que los tipos de datos que se almacenan o se recogen sean los adecuados.

```
import { Test, TestingModule } from '@nestjs/testing';
import { CategoriesService } from './categories.service';

describe('CategoriesService', () => {
  let service: CategoriesService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [CategoriesService],
    }).compile();

    service = module.get<CategoriesService>(CategoriesService);
  });

  it('should be defined', () => {
    expect(service).toBeDefined();
  });
});
```

Figura 60: Clase Service

Para la devolución de los datos están los archivos *.service*, que piden al controlador lo necesario y devuelven la vista deseada.

Los archivos *.module* son los encargados de dotar de funcionalidad general a todo este bloque, es decir, es el orquestador para el Modelo-Vista-Controlador ya que contiene los enlaces a los archivos que necesita esta carpeta y las rutas para este patrón.

Además de dentro de estas carpetas, la aplicación general contiene los mismos archivos para poder funcionar.

A parte de estos archivos, existen otras carpetas y archivos para conseguir la funcionalidad deseada.

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { PassportModule } from '@nestjs/passport';
import { UsersModule } from 'src/auth/users/users.module';
import { CategoriesController } from './categories.controller';
import { CategorySchema } from './categories.model';
import { CategoriesService } from './categories.service';

@Module({
  imports: [
    MongooseModule.forFeature([{
      name: 'Categories', // Collection
      schema: CategorySchema,
    }], 'apitfg'),
    PassportModule,
    UsersModule
  ],
  controllers: [CategoriesController],
  providers: [CategoriesService]
})
export class CategoriesModule {}
```

Figura 61: Clase Module

```
import { Module } from '@nestjs/common';
import { MongooseModule } from "@nestjs/mongoose";

@Module({
    imports: [
        MongooseModule.forRootAsync({
            connectionName: 'apitfg',
            useFactory: () => ({
                uri: 'mongodb+srv://sotoUmaTFG:znxHrBHcpWyDptMq@cluster0.cwrl9.mongodb.net/tfg?retryWrites=true&w=majority',
                useUnifiedTopology: true,
                useNewUrlParser: true,
            }),
        }),
    ]
})
export class MongoModule {}
```

Figura 62: Clase Module para MongoDB

Un ejemplo de ello es la carpeta database/mongo, encargada de gestionar la conexión con la base de datos de MongoDB que está en la nube. En su archivo mongo.module.ts se ejecuta la conexión con la base de datos.

También existen otros archivos de configuración para poder enlazar con las librerías que usa este proyecto, e incluso añadir otras librerías si fuera necesario.

Figura 63: Dependencias necesarias

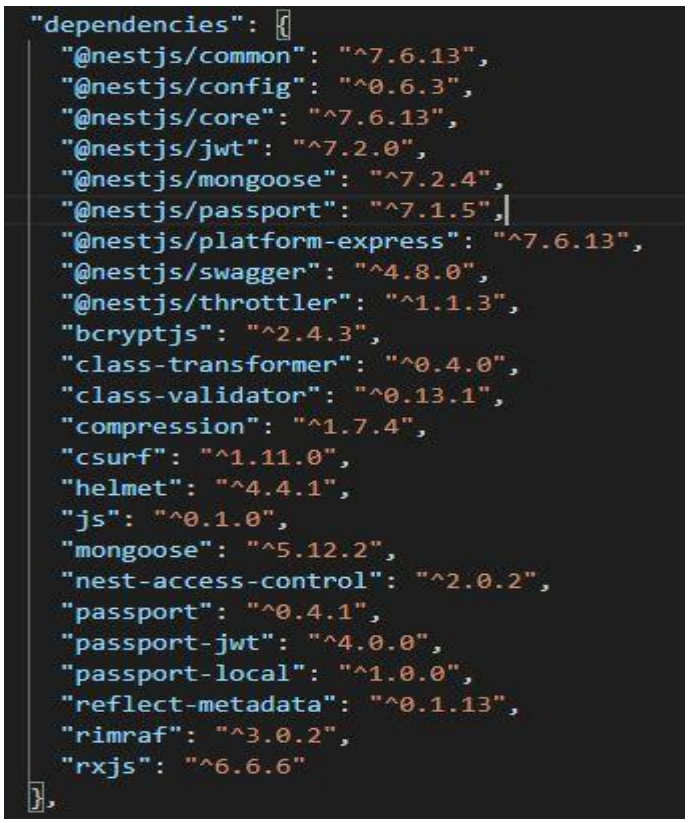


Figura 64: Dependencias para entorno de desarrollo

En el archivo package.json incluimos las librerías que se deben descargar tanto en el entorno de desarrollo como en el de producción, e incluso para poder hacer distintas pruebas o test.

4.2.3. DigitalOcean

Para la subida de archivos se ha optado por poder subirlos a la nube usando un servicio externo que ayuda con el peso en la aplicación. Tras un registro en el cloud de DigitalOcean, se puede mediante un par de peticiones post, subir o eliminar imágenes a su nube. En los datos de la publicación habrá un enlace a la imagen que el usuario suba en el momento de crear la publicación, creando así una consistencia entre la imagen y los datos de la publicación creada por el usuario. Para ello se han añadido constantes con los datos necesarios en el archivo *.env*, además de los siguientes archivos de imagen:

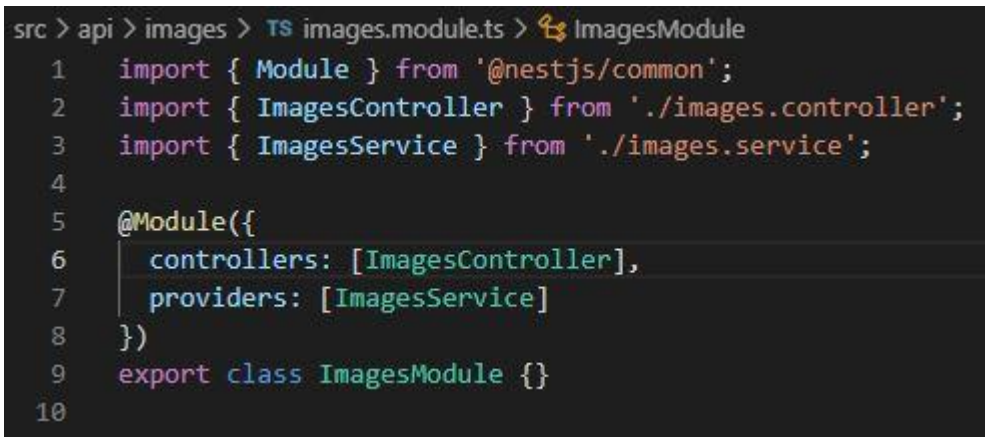


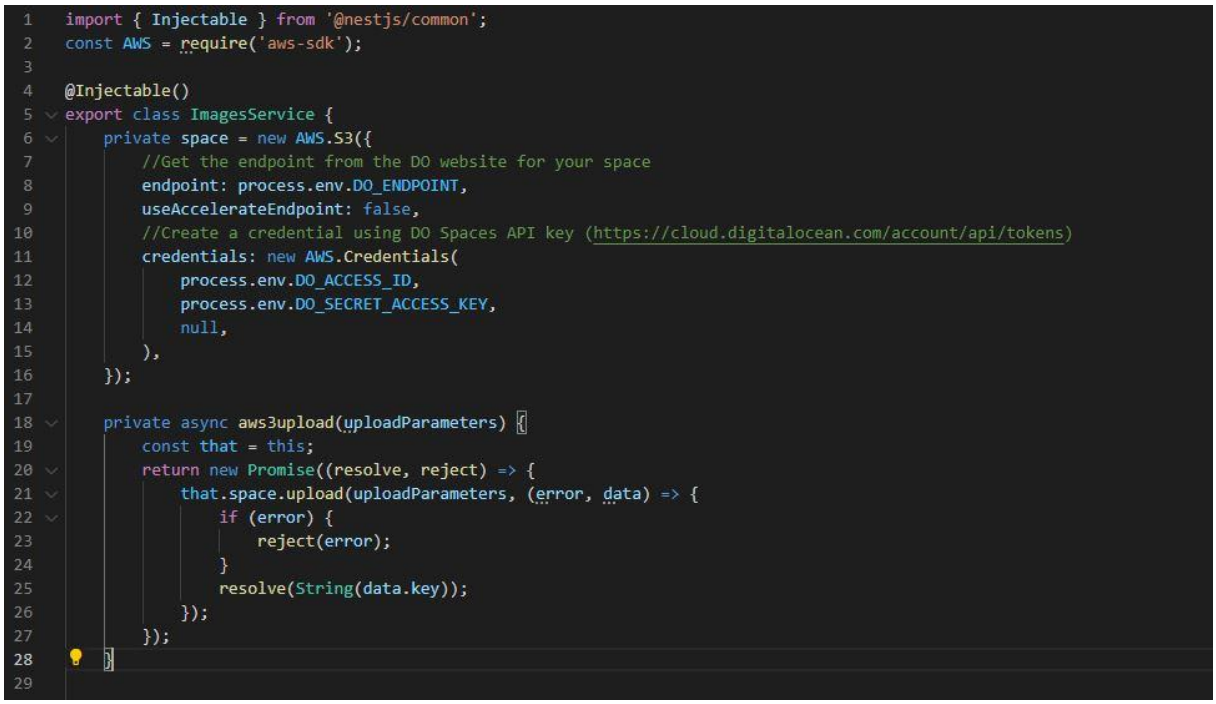
Figura 65: Módulo para la gestión de imágenes

```
import { Controller, Delete, Post, Query, UploadedFile, UseInterceptors } from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
import { ImagesService } from './images.service';

@Controller('images')
export class ImagesController {
    constructor(private imagesService: ImagesService) { }

    @Post('')
    @UseInterceptors(FileInterceptor('image'))
    async upload(@UploadedFile() file) {
        const pathAndFileName = await this.imagesService.upload(file);
        return pathAndFileName;

    }

    @Delete('')
    async delete(@Query('key') key: string) {
        const splited = key.split('/');
        const n = await this.imagesService.delete(
            splited[splited.length - 1],
        );
        return n;
    }
}

```

Figura 66: Controlador para la gestión de imágenes

Gracias a este archivo controlador se generan las rutas a la aplicación servidor para poder subir o eliminar imágenes.

En las figuras 67 y 68 se puede ver el código para ejecutar la subida y el borrado de las imágenes. Para ello se usa una librería *JS* que tras coger los datos de sesión de *DigitalOcean* permite hacer una conexión segura para usar los métodos necesarios para los mencionados objetivos.



```
import { Injectable } from '@nestjs/common';
const AWS = require('aws-sdk');

@Injectable()
export class ImagesService {
    private space = new AWS.S3({
        //Get the endpoint from the DO website for your space
        endpoint: process.env.DO_ENDPOINT,
        useAccelerateEndpoint: false,
        //Create a credential using DO Spaces API key (https://cloud.digitalocean.com/account/api/tokens)
        credentials: new AWS.Credentials(
            process.env.DO_ACCESS_ID,
            process.env.DO_SECRET_ACCESS_KEY,
            null,
        ),
    });

    private async aws3upload(uploadParameters) {
        const that = this;
        return new Promise((resolve, reject) => {
            that.space.upload(uploadParameters, (error, data) => {
                if (error) {
                    reject(error);
                }
                resolve(String(data.key));
            });
        });
    }

```

Figura 67: Servicio para la gestión de imágenes

```

    private async aws3delete(params) {
        return new Promise((resolve, reject) => {
            this.space.deleteObject(params, function (error, data) {
                console.log(data);

                if (error) {
                    reject(error);
                } else {
                    resolve(true);
                }
            });
        });
    }
    upload(file) {
        let uploadParameters = {
            Bucket: process.env.DO_BUCKET_NAME,
            ContentType: file.mimetype,
            Body: file.buffer,
            ACL: process.env.DO_UPLOAD_ACL, // ACL must be capitalized
            Key: `${file.originalname}`, //Key: req.file.file_name
        };

        const key = this.aws3upload(uploadParameters);
        return key;
    }

    async delete(fileName: string) {
        const params = {
            Bucket: process.env.DO_BUCKET_NAME,
            Key: `${fileName}`,
        };
        return await this.aws3delete(params);
    }
}

```

Figura 68: Servicio para la gestión de imágenes

4.2.4. Postman

Postman es una aplicación cliente HTTP que permite realizar peticiones a una API. A través de una interfaz gráfica de usuario (gui) se pueden realizar todo tipo de peticiones get, post, update o delete. Los datos son mostrados en diferentes formatos según queramos.

Para probar el funcionamiento de la aplicación servidor, se ha usado esta aplicación.

Se pueden ver distintas respuestas según las peticiones hechas a esta aplicación.

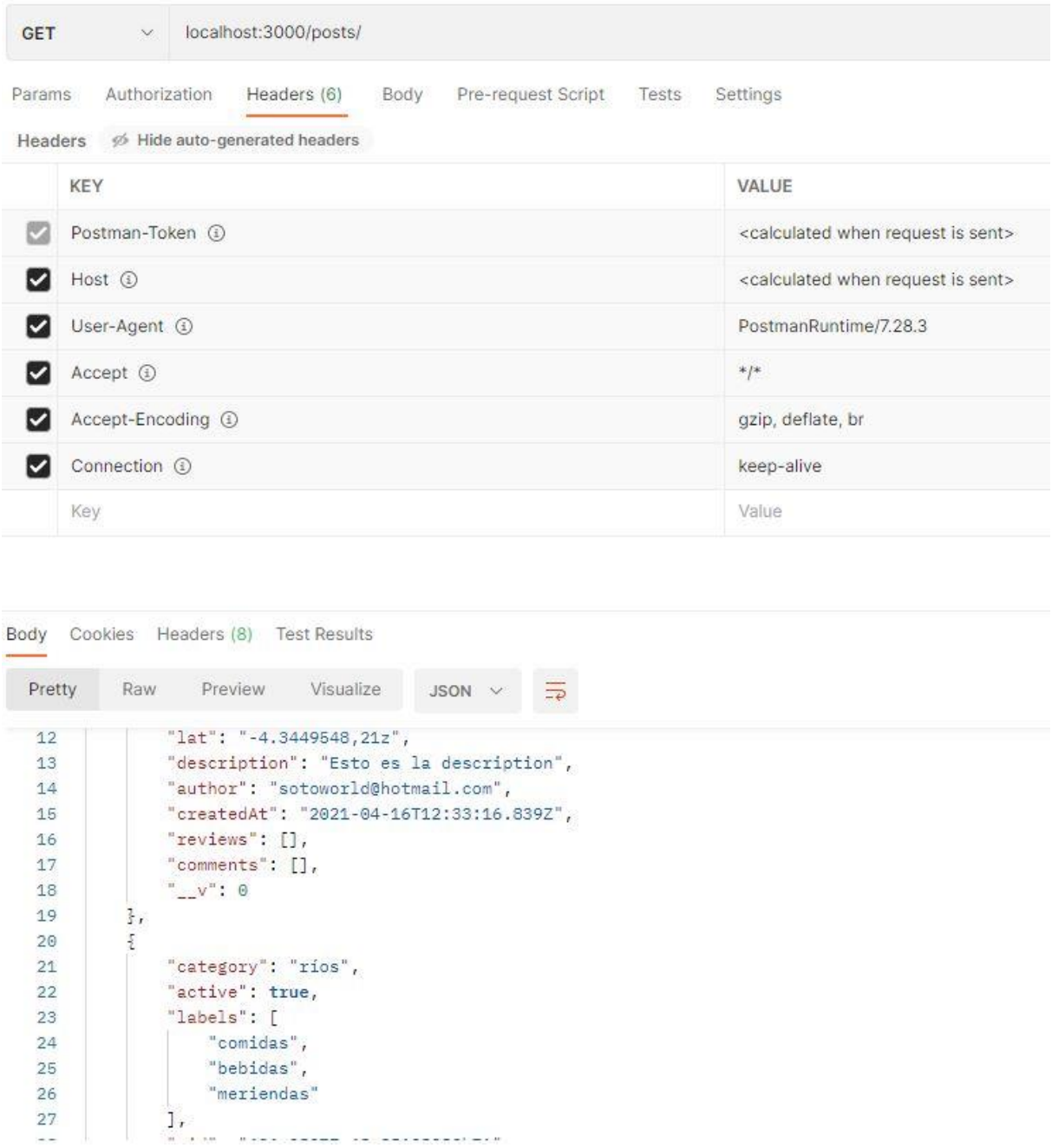


Figura 69: Ejemplo de método Get en Postman

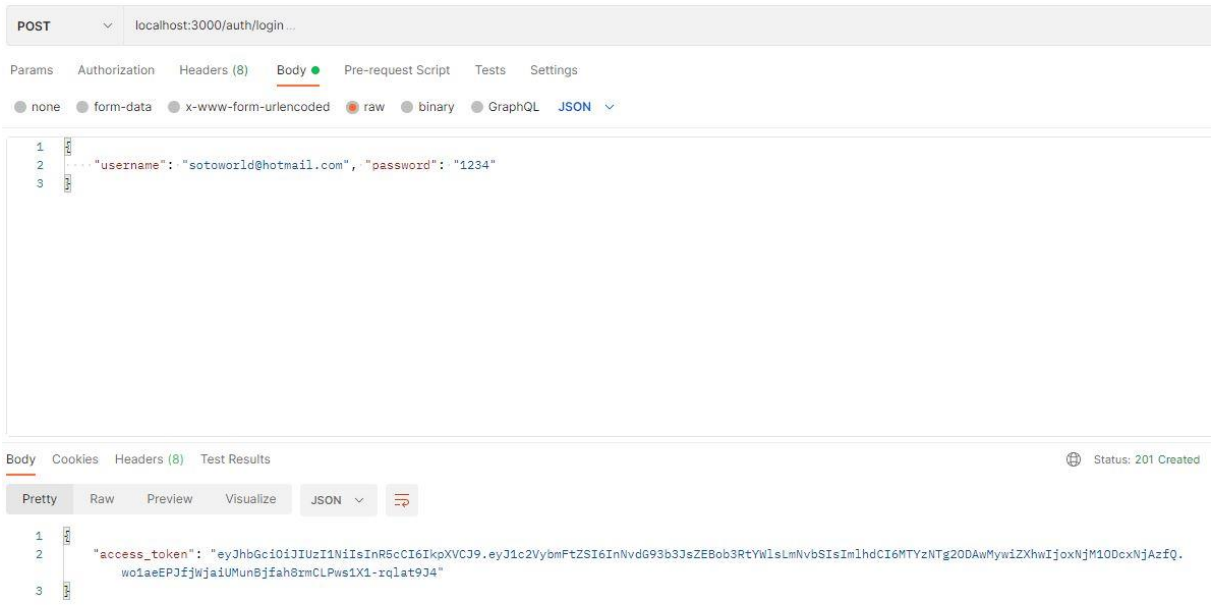


Figura 70: Ejemplo login Postman con método Post. Devolución de Token.

En este proyecto se ha usado Postman para poder realizar todas las pruebas pertinentes en el servidor y así verificar que funciona adecuadamente.

Con las figuras 69, 70 y 71 se pueden ver los resultados de respuesta en formato JSON que tendría que devolver la API.

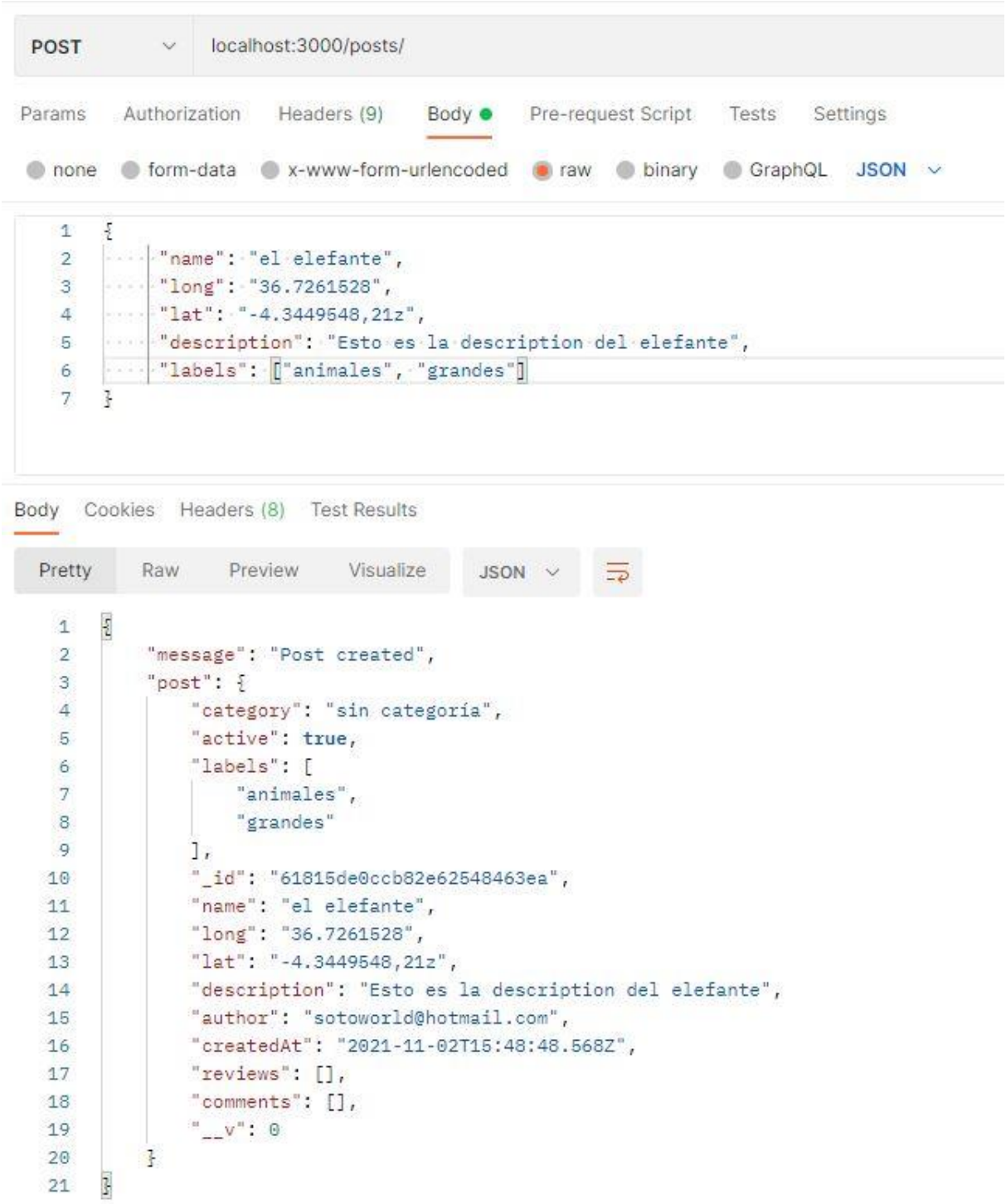


Figura 71: Ejemplo de Post en Postman para crear una nueva categoría.

Para no cansar al lector con tanta imagen no se añaden más figuras del trabajo en Postman, aunque conjuntamente con los archivos del proyecto se adjunta todo un repertorio de peticiones realizadas en Postman para verificar su correcto funcionamiento.

4.3. Implementación Cliente

4.3.1. Angular y Angular Material

Angular es un marco de diseño de aplicaciones y una plataforma de desarrollo para crear aplicaciones eficientes y sofisticadas de una sola página. Angular trabaja con componentes, elementos reutilizables compuestos a su vez por cuatro archivos. Estos archivos son el archivo *component.ts*, que contiene toda la lógica del componente. Un archivo *component.html* que implementa el *pseudo-código HTML* para la vista del componente. Un archivo *component.css* en el que se pueden incluir los estilos necesarios para modificar la visualización del componente elegido en el *HTML*. Y un archivo *component.spec* para realizar los test que se requieran.

Angular Material es una librería de estilos realizada por el equipo de Angular para integrarse perfectamente con Angular. Esta librería permite implementar componentes predefinidos basados en un diseño específico.

Para instalar Angular en nuestro ordenador se necesita de nuevo usar la consola y ejecutar el comando correspondiente, en este caso *npm install -g @angular/cli*.
Una vez instaladas las dependencias de Angular se pueden usar sus comandos para crear el proyecto, generar archivos y todo lo necesario para poder ejecutar la aplicación.Todos los comandos de Angular comienzan por *ng*.

Si ejecutamos *ng new nombre* se creará un nuevo proyecto Angular con el nombre elegido. En esta carpeta se habrán creado todos los archivos necesarios para empezar a programar la aplicación cliente. Por defecto Angular se ejecuta en el puerto 4200 (aunque se puede modificar si se requiere), por lo que al entrar a localhost:4200 podremos ver cómo está funcionando la aplicación tras ejecutar en la consola *ng serve*, comando encargado de arrancar la aplicación. Para comprobar que funciona se cargará una página de prueba que Angular trae incorporada y se visualizará al entrar en el navegador con esa URL.

Al igual que Nest, Angular comprende una serie de comandos que generan los archivos deseados para programar la aplicación:

- *ng g module nombre* para generar el módulo que se deseé. Creará una carpeta con el nombre asignado y un archivo que contendrá el módulo generado.
- *nest g component nombre* generará varios archivos de componente, uno con el componente *HTML*, uno con el *CSS*, y otro con el desarrollado en *Typescript*. Al ponerle el mismo nombre que el módulo lo ubicará en la misma carpeta.
- *ng g service services/nombre* generará el archivo servicio con la nomenclatura deseada en la ruta "*services*".

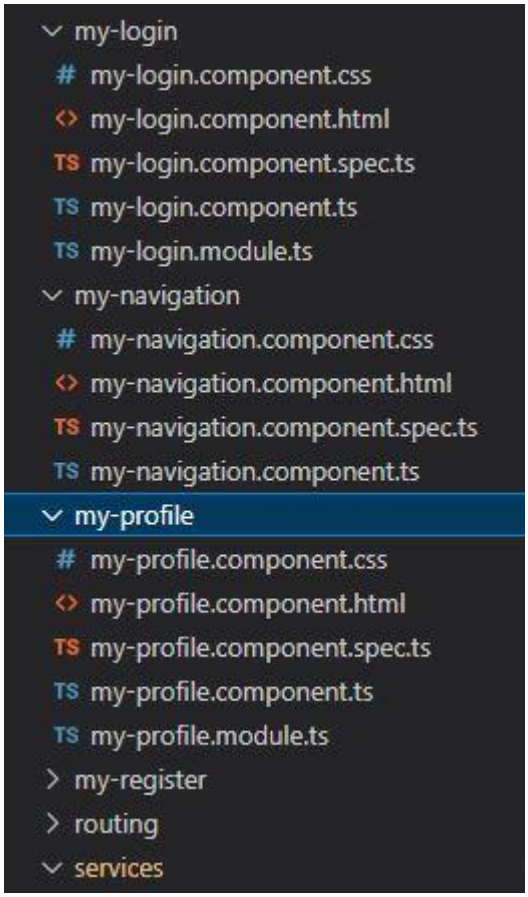


Figura 72: Ejemplo de estructura de archivos con sus componentes.

En la figura 72 se pueden observar ejemplos de estructuras para modelar un componente. Como se observa cada componente se separa de los demás por medio de una carpeta con el nombre de componente, como puede ser en este caso my-profile, my-navigation o my-login. Dentro de cada carpeta componente siempre tenemos un módulo con el nombre del componente, así como los 4 archivos componentes que se han generado a través del comando ejecutado para crear el componente.

```
import { Component } from '@angular/core';
import { inject } from '@angular/core/testing';
import { FormBuilder, Validators } from '@angular/forms';
import { AuthService } from '../services/auth.service';
import { MatDialogRef } from '@angular/material/dialog';


@Component({
  selector: 'app-my-login',
  templateUrl: './my-login.component.html',
  styleUrls: ['./my-login.component.css']
})
export class MyLoginComponent {
  loginForm = this.fb.group({
    username: [null, Validators.required],
    password: [null, Validators.required],
  });

  constructor(private dialogRef: MatDialogRef<MyLoginComponent>,private fb: FormBuilder, private authService:AuthService) {}

  onSubmit(): void {
    this.authService.login(this.loginForm.get('username')?.value, this.loginForm.get('password')?.value)
      .subscribe(result =>{ this.authService.accessToken = result.access_token;
        this.authService.accessToken ? this.dialogRef.close() : alert('Login incorrecto');
        localStorage["accesstoken"] = this.authService.accessToken;

      });
      //console.log(this.authService.accessToken);


  }
}
```

Figura 73: Ejemplo de la lógica del componente login.

En la figura 73 se observa la lógica del componente login. Este archivo escrito en TypeScript contiene toda la funcionalidad asociada a este componente. Se puede ver la lógica del formulario, así como su envío al servicio de autenticación.

```css
.dropdown-sub.dropdown-menu>div>span,
.megamenu-container ul li span {
    cursor: pointer;
    font-size: 15px;
    font-weight: bold;
    color: #16b1e9;
    margin-bottom: 5px;
}

.subsubmenu-header {
    display: flex;
}

.dropdown-sub.dropdown-menu>div>span:hover,
.megamenu-container ul li span:hover {
    color: #16b1e9;
}

.closebtn {
    font-size: 19px;
    width: 50px;
    display: flex;
    align-items: center;
    justify-content: center;
    margin-left: auto;
    cursor: pointer;
}
```

Figura 74: Ejemplo de código para la hoja de estilos en un componente.

En la figura 74 se ve un ejemplo de un archivo *CSS* en el que se inserta toda la lógica del diseño del componente. Mediante órdenes que se aplican a los elementos que pertenecen a la vista de la aplicación, se va dotando de orden, tamaño, colores y todo lo necesario para que cada elemento visualizado se pueda ver tal y como se quiere.

```
<form [formGroup]="loginForm" novalidate (ngSubmit)="onSubmit()">
  <mat-card class="shipping-card">
    <mat-card-header>
      <mat-card-title>Login</mat-card-title>
    </mat-card-header>
    <mat-card-content>
      <div class="row">
        <div class="col">
          <mat-form-field class="full-width">
            <input matInput placeholder="Username" formControlName="username">
          </mat-form-field>
        </div>
      </div>
      <div class="row">
        <div class="col">
          <mat-form-field class="full-width">
            <input type="password" matInput placeholder="Password" formControlName="password">
          </mat-form-field>
        </div>
      </div>
    </mat-card-content>
    <mat-card-actions>
      <button mat-raised-button color="primary" type="submit">Login</button>
    </mat-card-actions>
  </mat-card>
</form>
```

Figura 75: Ejemplo de archivo de html en el componente login.

La figura 75 muestra el elemento formulario para el componente login. Además de este elemento puede contener todos los que se quieran añadir. Este elemento está subcompuesto a su vez por otros elementos como los div, que son pequeñas cajas vacías para poder insertar dentro de ellas otros elementos, como son en este caso los inputs, cajitas para insertar datos como el usuario y la contraseña que se va a enviar al servidor.

```ts
import { waitForAsync, ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { NoopAnimationsModule } from '@angular/platform-browser/animations';
import { MatButtonModule } from '@angular/material/button';
import { MatCardModule } from '@angular/material/card';
import { MatInputModule } from '@angular/material/input';
import { MatRadioModule } from '@angular/material/radio';
import { MatSelectModule } from '@angular/material/select';

import { MyLoginComponent } from './my-login.component';

describe('MyLoginComponent', () => {
  let component: MyLoginComponent;
  let fixture: ComponentFixture<MyLoginComponent>;

  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [ MyLoginComponent ],
      imports: [
        NoopAnimationsModule,
        ReactiveFormsModule,
        MatButtonModule,
        MatCardModule,
        MatInputModule,
        MatRadioModule,
        MatSelectModule,
      ]
    }).compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(MyLoginComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should compile', () => {
    expect(component).toBeTruthy();
  });
});
```

Figura 76: Ejemplo de código para implementar las pruebas en el archivo my-login.component.spec.ts.

Como se ha mencionado con antelación, en los archivos .spec se añade la lógica necesaria para poder realizar pruebas en el código. Un ejemplo de ello es el que se puede ver en la figura 76 con verificaciones para el componente login.

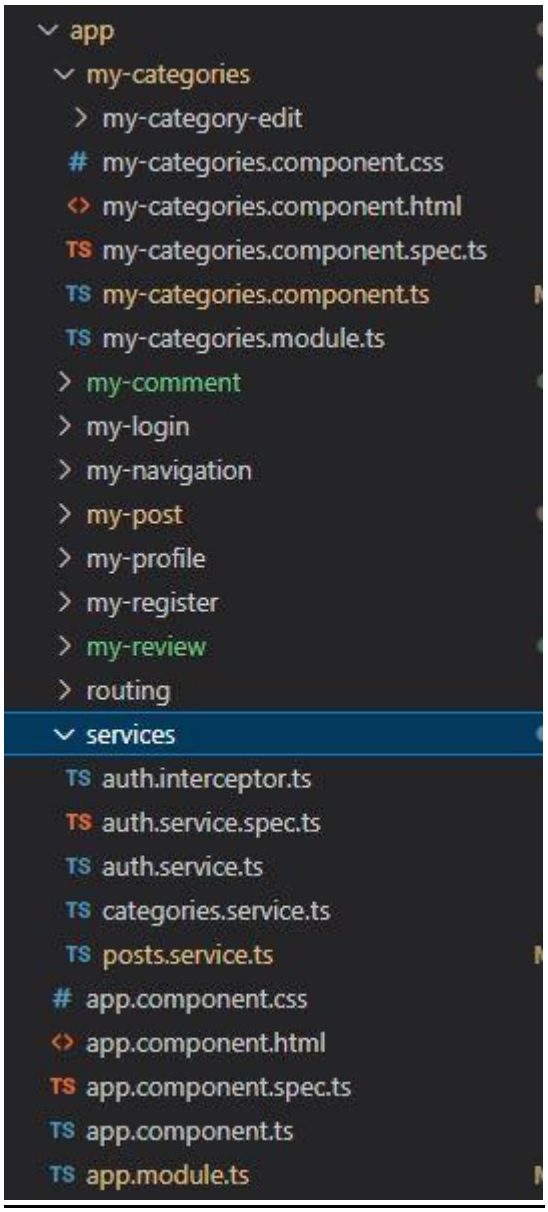


Figura 77: Árbol de archivos para la aplicación cliente.

En la figura 77 se muestra el árbol de archivos de la aplicación cliente, en él podemos ver como la lógica de la aplicación está separada en distintas carpetas con un nombre con el que se intuye para qué se usa dicha carpeta. En la carpeta my-categories se incluye una subcarpeta llamada my-category-edit con la que se aprecia que la edición de una categoría se hace a través de otro componente que se cargará en una pantalla modal (pop up) en la que aparecerá un formulario con los datos de la categoría en cuestión (figura número 78).

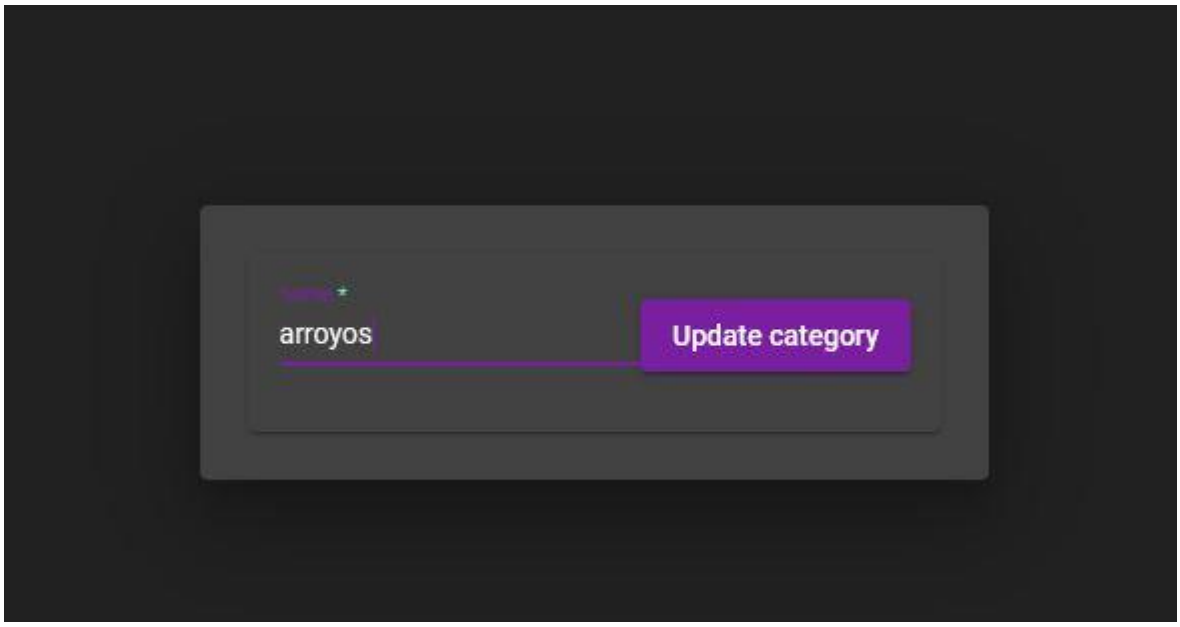


Figura 78: Vista de la edición de una categoría en un pop up

Antes de editar una categoría, se debe pulsar el botón de editar categoría que haya asociado a la categoría en cuestión. Estos botones para editar o eliminar una categoría, vienen en el listado de categorías (únicamente si el usuario que usa la aplicación cliente es de tipo administrador).

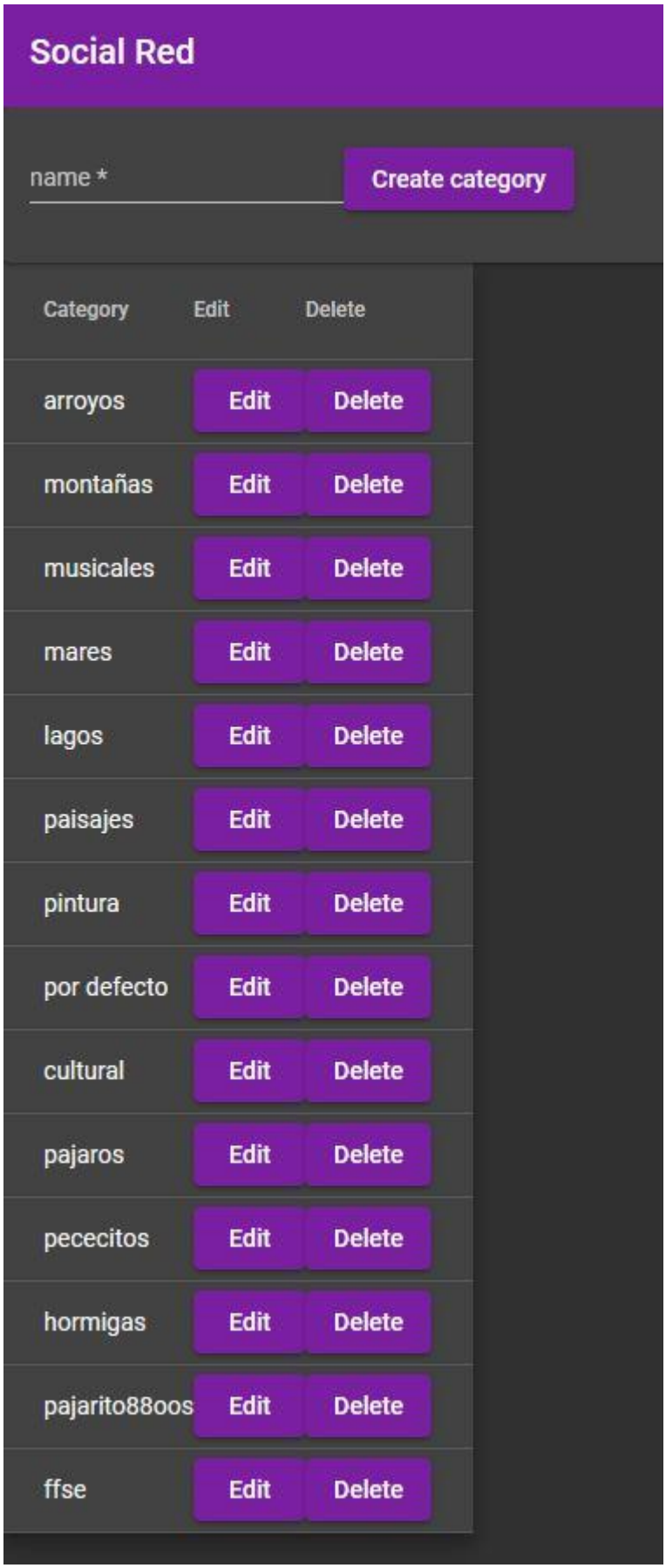


Figura79: Listado de categorías en la vista de la aplicación cliente por un usuario administrador

En la figura 79 aparecen varios ejemplos de categorías creadas que aparecerán al usuario a la hora de crear una publicación. Cada categoría tiene un botón asociado para editarla o eliminarla, tal y como se ve en la imagen. Estas categorías sólo pueden ser eliminadas o editadas por un usuario administrador.

Como se vio en el árbol de archivos (figura número 25), hay múltiples carpetas donde cada una contiene la lógica para un componente de la aplicación.

La carpeta my-login contiene los archivos necesarios para la lógica de inicio de sesión. En la figura 80 se muestra la vista del usuario para ingresar con su usuario y contraseña.

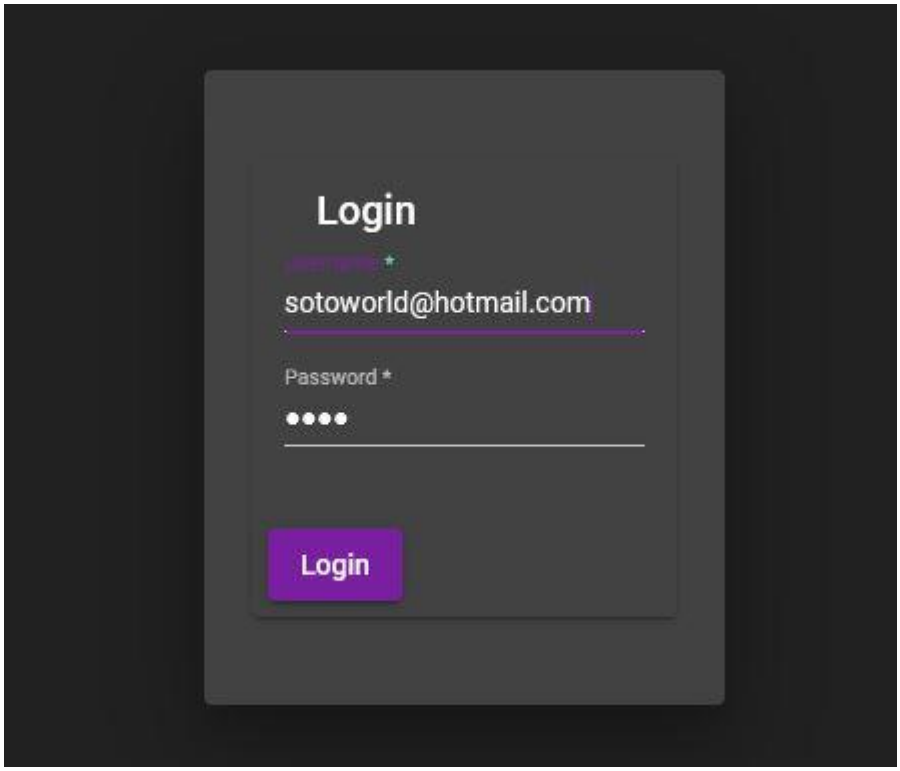


Figura 80: Formulario de inicio de sesión

Para acceder a todos estos componentes de la aplicación, el usuario puede pulsar los botones del menú que se muestran en la figura 81.

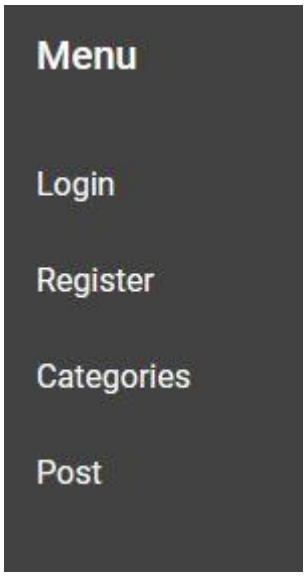


Figura81: Menú principal de la aplicación cliente

Tras iniciar sesión en la aplicación, el menú es modificado para que no aparezcan los apartados de login y register, y en su lugar aparezca el nombre del usuario para que pueda cerrar la sesión al pulsarlo.

En el menú de registro cualquier usuario puede registrarse en esta aplicación, introduciendo los datos pertinentes para el registro (nombre, correo, contraseña…). La carpeta my-register es la que contiene la lógica para hacer realidad la parte asociada a este registro.

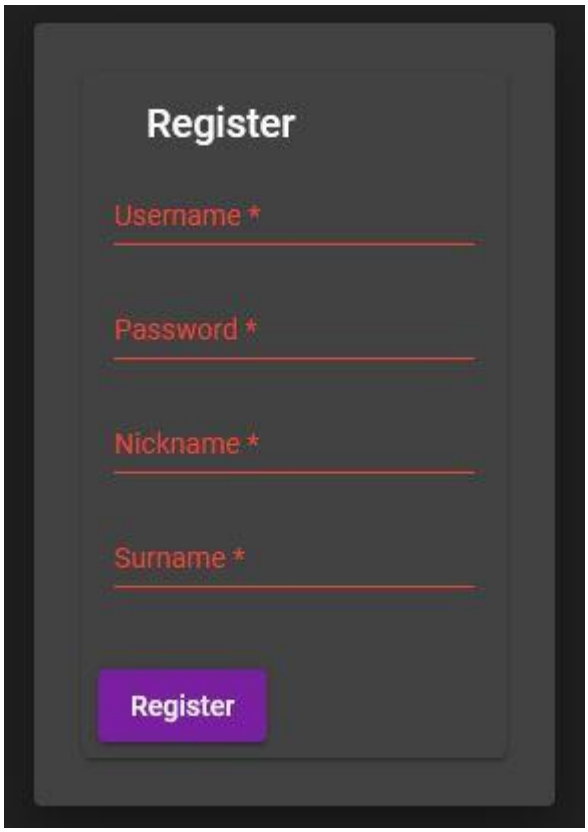


Figura 82: Formulario de registro

En la figura 82 se puede ver el formulario de registro que el usuario debe registrar para poder más tarde iniciar sesión.

En la figura 83 se muestra un listado de la vista de las categorías

Figura 83: Listado de publicaciones visto por un usuario administrador

En el listado de publicaciones de la figura número 83 se muestran todos los campos que puede contener una publicación.

Cada publicación tiene asociada una categoría, unas etiquetas, un nombre de publicación, una descripción, la fecha de creación de la publicación, así como la longitud y la latitud en el momento en que se subió la imagen al servidor. Además, se puede pulsar en el botón lateral comment o review para generar en un pop up el formulario correspondiente para crear un nuevo comentario o una puntuación nueva asociados a esa publicación.

Los botones Edit y Delete sólo se muestran en el listado si el usuario que accede es administrador. Como su propio nombre indican, sirven para editar o eliminar una publicación.

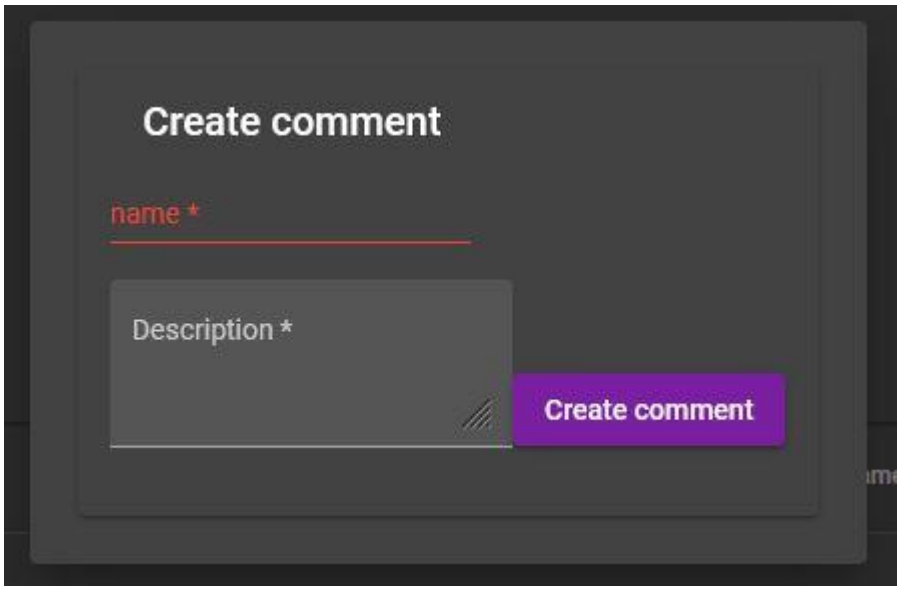


Figura 84: Pop up con el formulario para poder crear un nuevo comentario

En la figura 84 se muestra el formulario con el que se puede crear un comentario asociado a una publicación al pulsar en el botón *comment*. Para generar una nueva puntuación asociada a la publicación tan sólo se debe pulsar el botón *review* que mostrará un formulario para enviar la puntuación deseada (de 1 a 5).

Figura 85: Módulo con archivo routing

Para poder mapear las URLs por el que el usuario va navegando a través de la aplicación, se ha usado una clase de Angular que es capaz de renderizar las vistas asignadas a la URL que el usuario introduzca en la web. Este módulo se llama routing y sólo está compuesto por un archivo con el módulo de la aplicación para generar las rutas de la misma (el endpoint de la URL donde se requiere que se vea cada parte de la aplicación). En realidad, esta aplicación tiene pocas URLs para la parte cliente, ya que la mayoría de componentes se cargan en un popup, una ventana emergente que surge en la pantalla tras pulsar un botón o un enlace en la aplicación. En esta ventana emergente se cargan distintos componentes, la mayoría formularios con los que el usuario puede interactuar con el servidor a través de este componente.

En la figura 86 se observa el archivo módulo routing.module.ts con su código correspondiente.

src > app > routing > TS routing.module.ts > ...

```
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { MyCategoriesModule } from '../my-categories/my-categories.module';
4  import { Routes, RouterModule } from '@angular/router';
5  import { MyCategoriesComponent } from '../my-categories/my-categories.component';
6  import { MyPostComponent } from '../my-post/my-post.component';
7
8  const routes: Routes = [
9    { path: 'categories', component: MyCategoriesComponent },
10   { path: 'post', component: MyPostComponent },
11
12 ];
13
14 @NgModule({
15   declarations: [],
16   imports: [
17     CommonModule,
18     RouterModule.forRoot(routes)
19   ],
20   exports: [RouterModule]
21 })
22 export class RoutingModule { }
23
```

Figura 86: Código para rutear las urls de la aplicación cliente

Como hemos mencionado antes, además de las carpetas en las que hemos separado los archivos de la aplicación, como pueden ser la carpeta my-categories, o my-post, existe una carpeta llamada services que contiene los archivos .service.ts necesarios para realizar las peticiones externas que se hacen a la aplicación servidor.

Figura 87: Carpeta services con sus archivos

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})
export class CategoriesService {


  constructor(public http: HttpClient, public authservice: AuthService) { }

  private addTokenHeader():HttpHeaders{
    return new HttpHeaders({
      'Authorization': 'Bearer '+this.authservice.accessToken
    })
  }


  categories():Observable<any>{
    return this.http.get<any>('https://servertfg.herokuapp.com/categories');
  }

  newCategory(name:string):Observable<any>{
    return this.http.post<any>('https://servertfg.herokuapp.com/categories',{name } , {headers: this.addTokenHeader()})
  }

  editCategory(category:any):Observable<any>{
    return this.http.put<any>('https://servertfg.herokuapp.com/categories/'+category._id,{name:category.name } , {headers: this.addTokenHeader()})
  }

  deleteCategory(id:string):Observable<any>{
    return this.http.delete<any>('https://servertfg.herokuapp.com/categories/'+id , {headers: this.addTokenHeader()})
  }

  getCategory(id:string):Observable<any>{
    return this.http.get<any>('https://servertfg.herokuapp.com/categories/'+id)
  }
```

Figura 88: Ejemplo de archivo servicio: CategoriesService

En la figura 88 se puede ver un ejemplo de un archivo service. Este archivo contiene la lógica para conectar con la aplicación servidor. Gracias a este archivo se pueden hacer las distintas peticiones al servidor (get, post, put, delete) para listar, crear, editar o eliminar una categoría.

4.4. Subida de archivos a la nube

4.4.1. GitHub

Para el control de versiones tanto de la parte cliente como de la parte servidor de este proyecto se optó por usar GitHub.

Tras crear una cuenta en la plataforma asociada al correo electrónico del autor de este proyecto, se han creado dos proyectos que se han subido a GitHub.

Por un lado, se ha creado un repositorio en GitHub con el servidor donde se han ido subiendo los cambios que se han hecho en cada archivo.

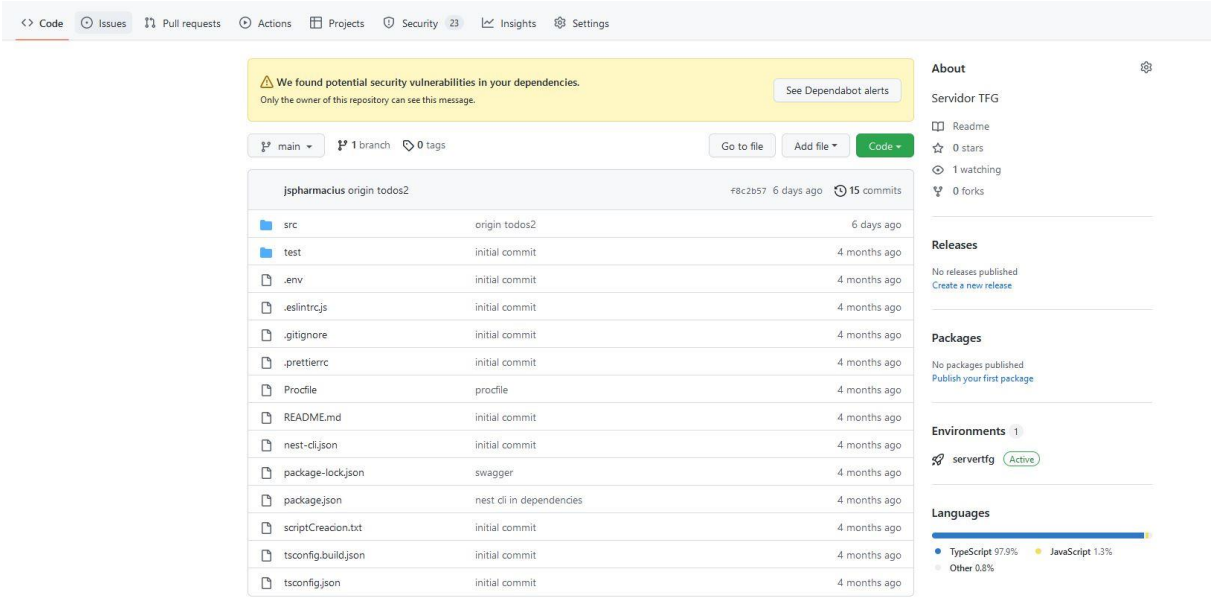


Figura 89: Git para el servidor

Para el cliente se ha hecho lo mismo, pudiendo así controlar las versiones de la aplicación conforme los archivos van cambiando.

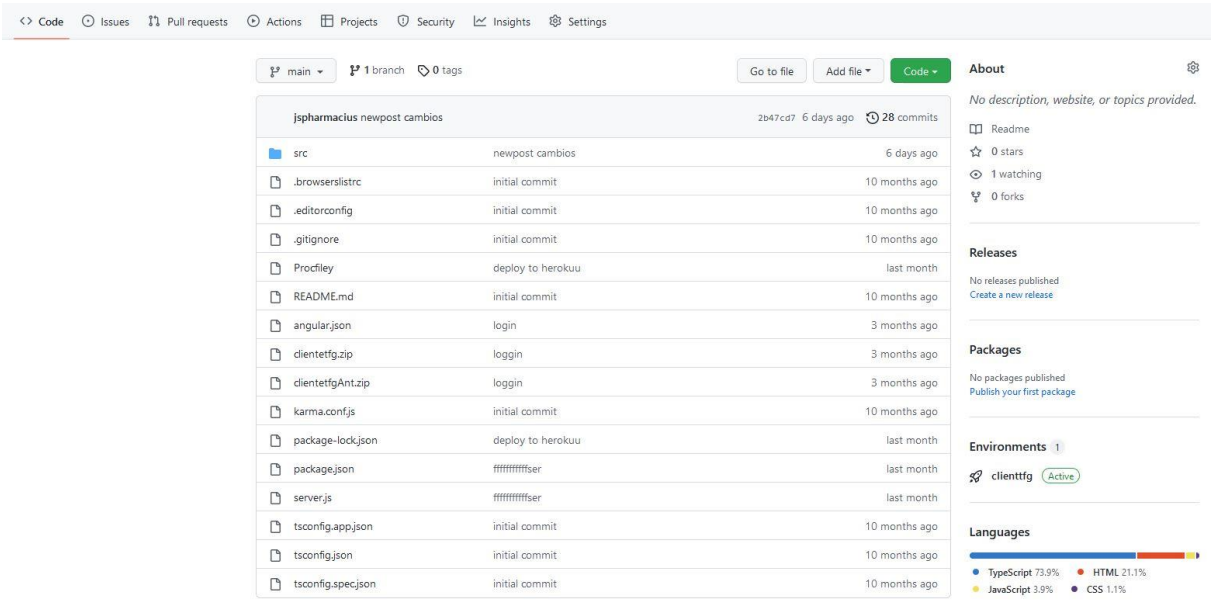


Figura 90: Git para el cliente

Gracias a *GitHub* se puede ver todo el historial de cambios tanto del cliente como del servidor, y volver a restaurar una versión antigua de la aplicación si fuera necesario. Además, se puede descargar cualquiera de los dos proyectos con un sencillo comando y poder probar la aplicación en local o en cualquier alojamiento. Mediante distintos comandos, y previo almacenar el usuario y la clave de *GitHub* en nuestro entorno local, podemos generar distintas ramas, eliminarlas, unirlas y descargar o subir los archivos al repositorio *Git*.

A continuación, se exponen una serie de comandos necesarios para trabajar con *Git*:

- *Git init nombreProyecto* crea un nuevo repositorio local con el nombre del proyecto indicado.
- *Git clone nombreRepositorio* crea un nuevo repositorio que es un clon del repositorio que se indica en nombreRepositorio.
- *Git config* permite establecer una configuración específica del usuario, como puede ser su nombre de usuario, email y distintos formatos.
- *Git add nombreArchivo* agrega el archivo indicado al área de trabajo de git. Así está listo para la siguiente operación. Si en vez del nombre del archivo se pone un asterisco, se agregarán todos los archivos que se hayan modificado después del último add ejecutado.

- *Git commit* captura una instantánea de los cambios preparados mediante *git add* en ese momento del proyecto. Esta instantánea se puede considerar como una versión segura del proyecto, y se almacenará en la versión de cambios de *Git*.
- *Git status* muestra información acerca de la lista de los archivos modificados, preparados o confirmados.
- *Git branch* se usa para listar, crear o borrar ramas. Este comando permite distintos parámetros para poder usar la acción que se quiere realizar o poder elegir el nombre de la rama. Una rama es simplemente un puntero móvil que apunta a una de las confirmaciones mencionadas con antelación. Esto es muy útil cuando en un mismo proyecto trabajan varias personas, ya que cada una puede crear una rama distinta y trabajar en la misma. A la hora de unir las ramas si hubiera algún conflicto (por ejemplo, que un usuario hubiera modificado el mismo archivo que otro usuario) git avisaría para poder actuar en consecuencia. La rama principal de un repositorio git se llama rama master.
- *Git merge* permite unir las ramas creadas con *git branch*.
- *Git pull* descarga y fusiona los datos del repositorio remoto con los del repositorio local.
- *Git push origin master* sube y fusiona los datos del repositorio local con los del repositorio remoto en la rama master. Es la opuesta de *git pull*.

Hay otros muchísimos más comandos que se han usado en este proyecto o se pueden usar en git, pero no vamos a listarlos todos para no sobrecargar al lector.

Para alojar este proyecto se decidió usar *Heroku*, por lo que usamos git para desplegar en esta nube los dos repositorios que se han creado en *GitHub*.

4.4.2. Heroku

Como se adelantó en la introducción, para subir este proyecto se usará una plataforma como servicio (PaaS) llamada Heroku.

Primero se deben crear las instancias de cada proyecto que se quiera subir a Heroku.

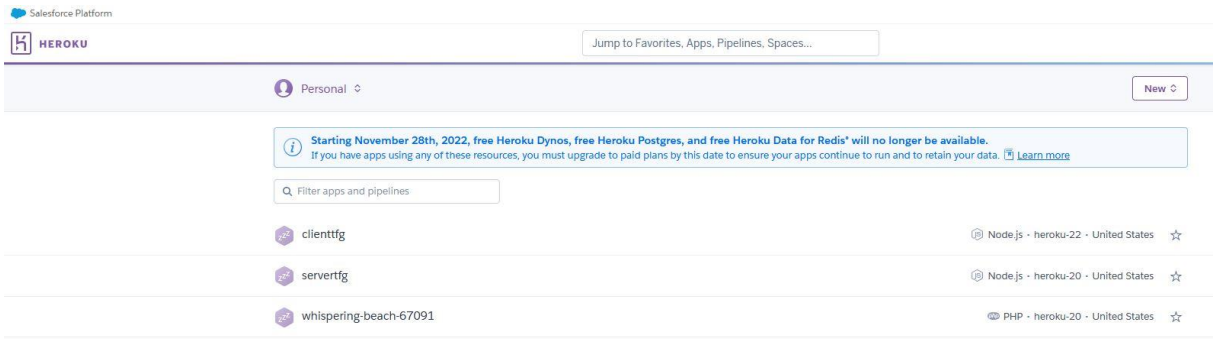


Figura 91: Distintas instancias creadas en heroku

En la figura 91 se pueden ver tres instancias creadas en Heroku.

La de abajo el todo (whispering) es una prueba en lenguaje PHP para ver y probar el funcionamiento de esta plataforma.

Servertfg es la instancia que contiene todos los archivos de la aplicación servidor para este proyecto, y clientetfg es la instancia que contiene todos los archivos de la aplicación cliente para este proyecto.

Una vez creadas las instancias del cliente y el servidor en Heroku, se puede enlazar fácilmente los repositorios de GitHub a estas instancias. Con unos cuantos clicks y entrando a GitHub con el usuario y contraseña de la plataforma, Heroku se conecta a los distintos repositorios indicados.

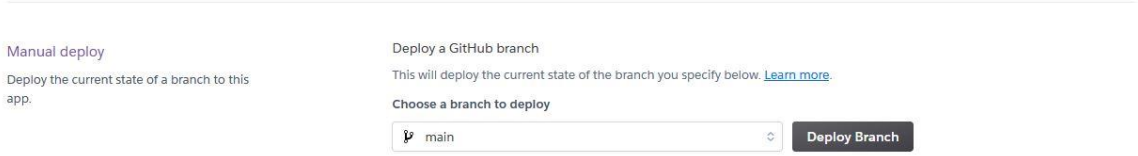


Figura 92: Deploy manual en Heroku

Para subir los archivos a Heroku desde GitHub (deploy) tan sólo es necesario pulsar un botón tal y como se puede ver en la figura número 92. Cada instancia (la del cliente y la del servidor) está ligada mediante el paso previo explicado a su correspondiente repositorio. En cada una de estas instancias se puede pulsar el botón Deploy Branch para subir los archivos y desplegar la aplicación en la nube para que el usuario pueda acceder mediante las URLs correspondiente.

https://clienttfg.herokuapp.com/ es la URL de entrada para los usuarios. A través de esta URL se puede acceder a visualizar y usar la aplicación definida en este documento.

https://servertfg.herokuapp.com/ es la URL de entrada al servidor de datos de esta aplicación. El cliente puede lanzar peticiones de distinto tipo (post, get, put…) contra esta URL para realizar las acciones que necesite y así poder recoger, insertar, modificar o eliminar datos. En el punto 3.2.4 (Postman) explicado en esta memoria, se usa esta URL para lanzar las peticiones y recoger los datos que se han visto en las distintas imágenes.

Además de todas estas opciones, Heroku también permite visualizar un registro de logs para poder observar los disntintos problemas o estados por los que va pasando la aplicación cuando se hace un deploy.

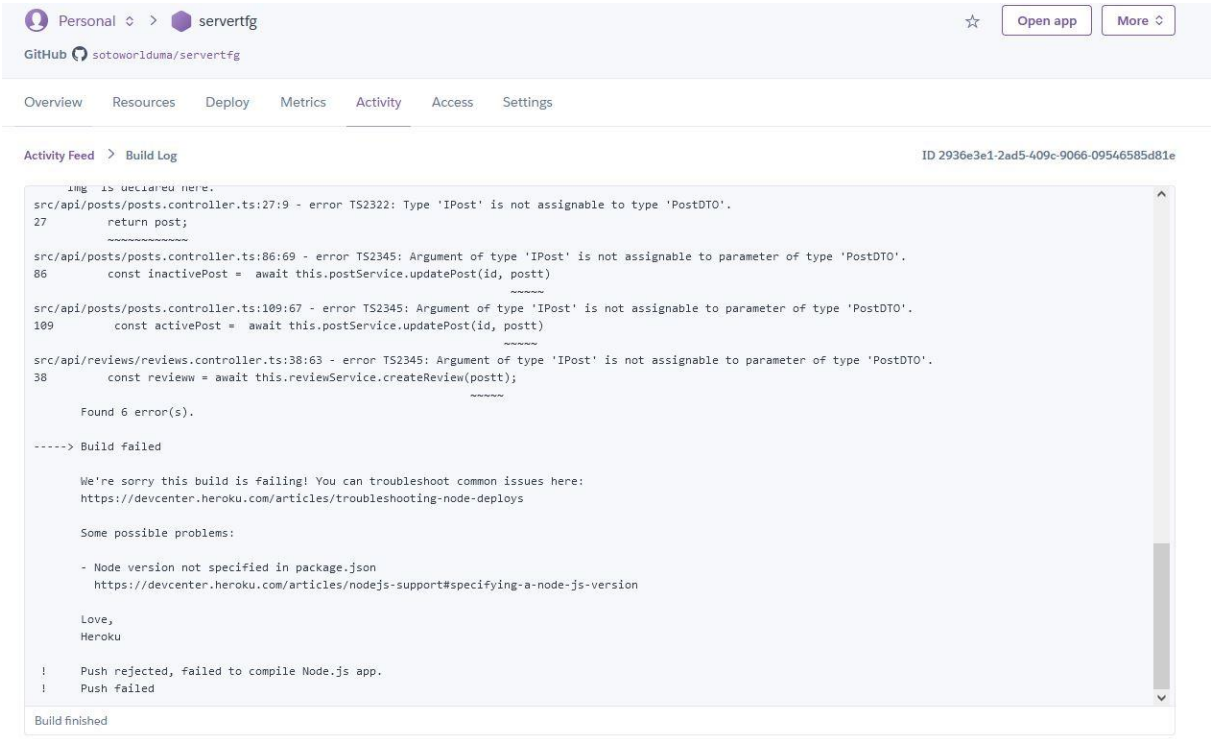


Figura 93: Deploy manual en Heroku

En la figura 93 se ve un log de un deploy en Heroku tras recoger los datos del repositorio GitHub.

Además, se puede ver un registro de actividad con las distintas subidas y si alguna tuvo algún error al hacer el deploy. Se puede ver este registro en la figura 30.

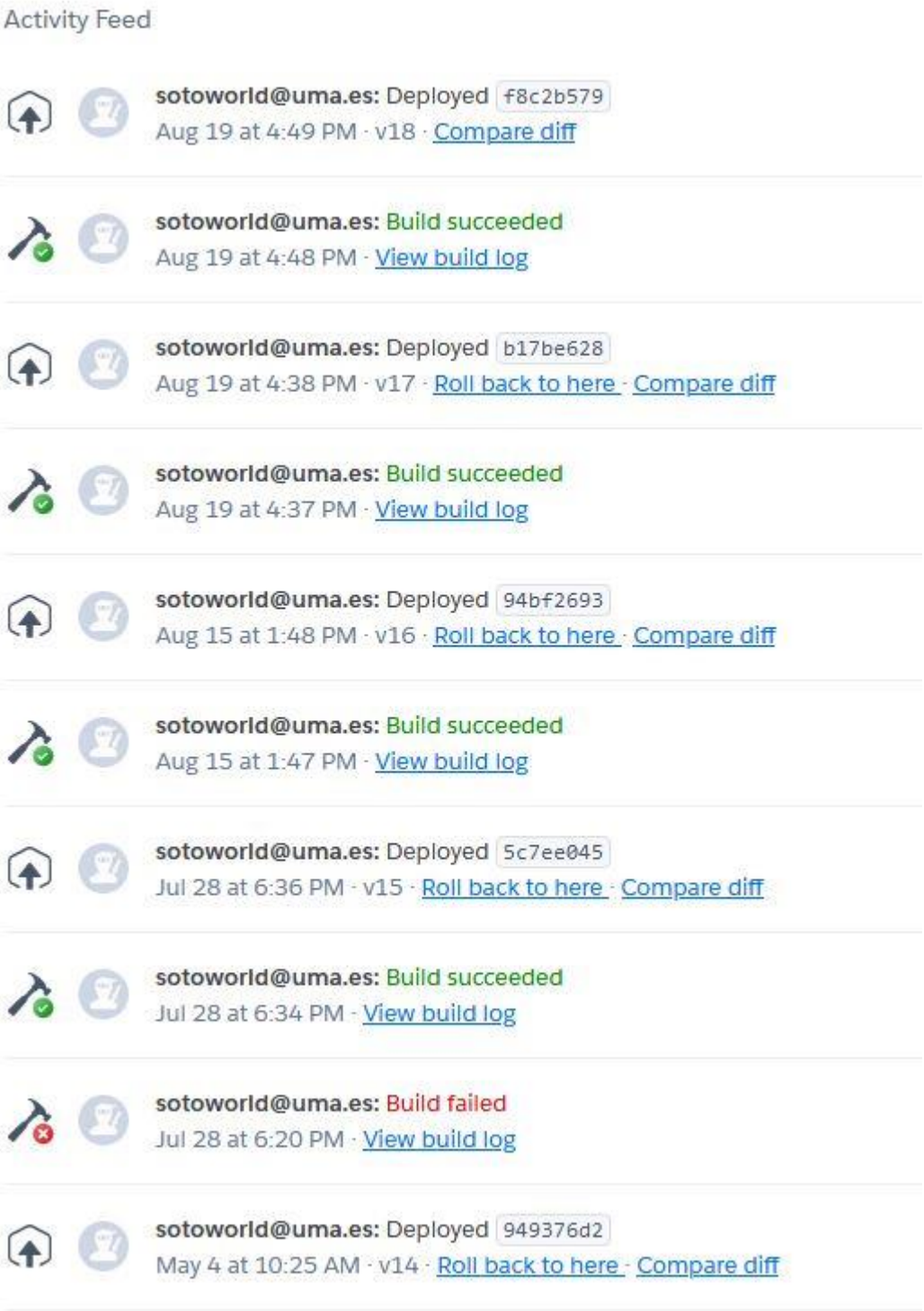


Figura 94: Registro de logs de distintos deploy

5

Conclusiones y mejoras futuras

En este capítulo se expondrán las conclusiones a las que se ha llegado tras la creación de este proyecto viendo los objetivos logrados, la problemática que se ha encontrado en la realización, así como las posibles mejoras en las funcionalidades o requisitos.

5.1 Conclusiones

Como objetivo en este proyecto se planteo crear una red social para lugares de interés con el fin de facilitar al usuario una aplicación en la que pudiera subir imágenes de cualquier lugar y en cualquier sitio. Esta aplicación ayudaría a otros usuarios a saber dónde se tomaron estas imágenes o fotografías y así poder ir a visitarlas en persona si el usuario así lo quisiera.
La elección de las tecnologías utilizadas tiene sus pros y sus contras. Como pro se puede destacar que son tecnologías de última generación, muy rápidas y escalables, aunque también tienen sus contras. Precisamente ser de última generación es lo que ha llevado al autor de este proyecto a utilizarlas, ya que, aunque tenía más experiencia en otras tecnologías, siempre viene bien aprender lenguajes nuevos y modernos. Al ser tan modernas tienen total compatibilidad con las plataformas web más modernas, por lo que se ha podido subir a Heroku este proyecto para que tenga una visibilidad total.

Durante la creación del proyecto se ha aprendido muchísimo sobre estas tecnologías, y sobre como gestionar un proyecto desde cero, pasando por todas las fases necesarias, desde la creación de requisitos por parte del cliente hasta la generación de la aplicación cliente y servidor pasando por todas las fases intermedias que se han plasmado en este proyecto. Para ello se usó una metodología ágil pudiendo implementar el proyecto en los distintos sprint que se han explicado en el capítulo 2.

Al final se ha podido implementar correctamente el proyecto, funcionando correctamente la aplicación cliente y la aplicación servidor, aunque si que ha surgido bastante problemática, sobre todo en la aplicación cliente.

La creación de este proyecto ayuda muchísimo a verificar que se puede hacer frente a la problemática que vaya surgiendo a la hora de trabajar y uno siente que ha pasado en mayor o menor medida por todas las fases que intervienen en la creación de cualquier aplicación.

Sin duda es muy recomendable realizar un proyecto de este tipo al finalizar los estudios.

5.2 Problemáticas en la ejecución del proyecto

A lo largo de la ejecución de este proyecto han aparecido multitud de problemas.

Uno en particular todavía persiste a día de hoy. A la hora de hacer una petición para enviar la imagen que se necesita subir al servidor, el navegador informa de un problema de *CORS*.

Este problema de control de acceso en el intercambio de recursos de origen cruzado todavía no ha podido se resuelto. No existe demasiada documentación para este problema en los foros que se han visitado. Se ha intentado implementar todas las soluciones que se han propuesto en estos mismos foros, aunque ninguna ha surtido efecto. Una de estas soluciones ha sido implementar un proxy en la aplicación cliente, que, aunque funciona perfectamente, no ha resultado como se esperaba.

Otra de las principales problemáticas ha sido aprender a usar un lenguaje y unos frameworks totalmente desconocidos (en términos de uso) para el autor, aunque gracias a que hay bastante documentación al respecto se ha podido hacer frente a este problema con bastante

solvencia. Sin embargo, existe mucha documentación con funcionalidades obsoletas o que no funcionan bien (siempre tutoriales externos a la documentación oficial).

También ha sido un desafío trabajar con *MongoDB* al ser la primera vez en usar una base de datos *NoSQL*.

Otro de los principales problemas ha surgido al subir el proyecto a la nube de *Heroku*. Al subir el servidor no hubo ningún problema, todo funcionaba correctamente, el problema vino al subir el cliente. Aunque en local funcionaba perfectamente, si se desea que el cliente funcione en *Heroku* hace falta un pequeño servidor que despliegue la aplicación cliente en el servidor web. Para ello se ha creado un pequeño servidor hecho con *Express,* tal y como se ve en la figura 96. Este pequeño servidor permite que *Heroku* pueda servir la aplicación cliente correctamente. También se tuvieron que hacer múltiples pruebas de configuración hasta dar con la tecla para que *Heroku* mostrara correctamente la aplicación cliente.

```
const path = require('path');
const express = require('express');
const app = express();

// Serve static files
app.use(express.static(__dirname + '/dist/clientetfg'));

// Send all requests to index.html
app.get('/*', function(req, res) {
  res.sendFile(path.join(__dirname + '/dist/clientetfg/index.html'));
});

// default Heroku port
app.listen(process.env.PORT || 8080);
```

Figura 96: Servidor Express.

5.3 Mejoras futuras

A una red social siempre se le pueden añadir nuevas funcionalidades que alegrarían a los usuarios de la aplicación.

Una posible mejora sería un canal chat de punto a punto, donde el usuario pudiera hacer una petición de chat a otro usuario y tras aceptarla, ambos pudieran chatear a través de la red social. Esto crearía más dinamismo a la aplicación, dónde seguro haría que más usuarios quisieran usarla.

Otra posible mejora sería aumentar la calidad de las imágenes que se subieran al servidor, así como poner una marca de agua y así saber que proviene de la aplicación si alguien las descarga. Existen distintas librerías que pueden mejorar la calidad de la imagen para que sea más "apetecible" al usuario.

También relacionado con la calidad o la marca de agua, sería interesante añadir unos filtros para las imágenes, que pueden hacer que la imagen sea más curiosa. En determinadas circunstancias, un filtro puede ayudar a que la imagen sea más vistosa o agradable.

Algo interesante sería el poder añadir un hilo musical a una foto. Muchas redes sociales ya permiten esta mejora, consiguiendo un punto emotivo al ver la imagen con el sonido de fondo.

Otra idea interesante sería un botón con el que compartir la publicación con otras personas. Al pulsar el botón los usuarios podrían elegir la forma de compartir el archivo, bien vía web, vía *WhatsApp* o *Telegram*, por correo, o incluso por otra red social. Esto enviaría un enlace con un texto a través del recurso elegido y así otras personas podrían enlazar con la publicación, aumentado a su vez el volumen de usuarios en la aplicación.

Algo que algunos usuarios agradecerían sería privacidad en su publicación. Se podría colocar un *check* para conseguir que la publicación que un usuario crea no sea pública. A través de un listado con otros usuarios se podría dar permisos de visualización al post, consiguiendo así que solamente pueda verlo los usuarios que el creador de la publicación quiera. Esto es un

doble rasero ya que así se limitarían las visitas, comentarios y valoraciones en esta publicación, por lo que en vez de aumentar las visitas podría reducirlas, pero algunos usuarios lo agradecerían.

Referencias

[1] Documentación Nest, 2022

Sitio web: https://docs.nestjs.com/

[2] Wikipedia, Atom, 2022

Sitio web: https://es.wikipedia.org/wiki/Atom_(software)

[3] UA, AJAX, 2022

Sitio web: https://si.ua.es/es/documentacion/mootools/ajax.html

[4]] Wikipedia, NodeJS, 2022

Sitio web: https://es.wikipedia.org/wiki/Node.js

[5]] Wikipedia, Express, 2022

Sitio web: https://en.wikipedia.org/wiki/Express.js

[6]] Wikipedia, MongoDB, 2022

Sitio web: https://es.wikipedia.org/wiki/MongoDB

[7]] Wikipedia, LESS, 2022

Sitio web: https://es.wikipedia.org/wiki/LESS_(lenguaje_de_hojas_de_estilo)

[8]] Wikipedia, Git, 2022

Sitio web: https://es.wikipedia.org/wiki/Git

[9]] Economipedia, HTML, 2022

Sitio web: https://economipedia.com/definiciones/lenguaje-html.html

[10]] Wikipedia, CSS, 2022

Sitio web: https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada

[11]] Arsys, arquitectura, 2022

Sitio web: https://www.arsys.es/blog/arquitectura-software

[12]] Wikipedia, Bootstrap, 2022

Sitio web: https://es.wikipedia.org/wiki/Bootstrap_(framework)

[13]] Aquíhaydominios, Awesome, 2022

Sitio web: https://www.aquihaydominios.com/blog/font-awesom-que-es-y-como-se-usa/

[14]] Gfourmis, NestJS, 2022

Sitio web: https://gfourmis.co/nestjs-que-es-y-por-que-empezar-a-usarlo/

[15]] Mongodb, MongoDB, 2022

Sitio web: https://cloud.mongodb.com

[16]] Qualitydevs, Angular, 2022

Sitio web: https://www.qualitydevs.com/2019/09/16/que-es-angular-y-para-que-sirve/

[17]] Angular, Angular, 2022

Sitio web: https://angular.io

[18]] Angular, Angular Material, 2022

Sitio web: https://material.angular.io/

[19]] Heroku, Heroku, 2022

Sitio web: https://heroku.com/

[20]] Youtube, varios videos, 2022

Sitio web: https://youtube.com/

[21]] Educative, Arquitecturas, 2022

Sitio web: https://www.educative.io/blog/how-to-design-a-web-application-software-architecture-101#what

[22]] Geeks, Arquitectura en capas, 2022

Sitio web: https://geeks.ms/jkpelaez/2009/05/30/arquitectura-basada-en-capas/#:~:text=La%20arquitectura%20basada%20en%20capas,funcionalidad%20que%20est%C3%A1%20siendo%20desarrollada.

[23]] Desarrolloweb, Nest, 2022

Sitio web: https://desarrolloweb.com/home/nestjs

[24]] Ramoncarrasco, MongoDB, 2022

Sitio web: https://www.ramoncarrasco.es/es/content/es/kb/129/el-nucleo-mongod-y-la-arquitectura-de-mongodb

[25]] Wikipedia, Modelo MVC, 2022

Sitio web: https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93modelo_de_vista

[26]] Adictosaltrabajo, Arquitecturas, 2022

Sitio web: https://www.adictosaltrabajo.com/2012/10/07/zk-mvc-mvvm/#:~:text=Comparaci%C3%B3n%20entre%20MVC%20y%20MVVM,-

La%20primera%20imagen&text=Las%20principales%20diferencias%20entre%20MVC,Controller%C2%BB%20como%20sucede%20en%20MVC.

Apéndices

Apéndice A

Manual de Instalación del servidor

Requerimientos:

Para poder instalar tanto el cliente como el servidor en local, es necesario tener instalado *NodeJS* y el gestor de paquetes *npm* (Node Package Management).

Para instalar *NodeJS* se entrará en la url https://nodejs.org/es/download/
Allí pulsaremos sobre LST (Recomendado para la mayoría) y pulsaremos sobre el archivo de Node.js que más nos convenga, según el sistema operativo donde queramos intalarlo.

Una vez descargado el archivo sólo es necesario hacer doble click en él y seguir los pasos de instalación dándole a siguiente hasta finalizar la instalación.

Al completar la instalación ya tendremos instalado *NodeJS*, que vendrá por defecto también con el instalador de paquetes *npm*.

Una vez instalados estos requerimientos ya es posible empezar con la instalación de los archivos de este proyecto. Si se quiere verificar la instalación de Node se puede ejecutar

por consola el comando *node -v*, que nos mostrará la versión instalada de *NodeJS*. Al ejecutar en consola *npm -v* también se mostrará la versión de *npm* que se ha instalado.

Instalación:

Este proyecto está compuesto por dos archivos comprimidos en formato ZIP. Cada uno de ellos se corresponde con la aplicación cliente y la aplicación servidor.

Para el servidor descomprimiremos el archivo servertfg. Al descomprimir se verá una carpeta servertg que abarca todos los archivos esenciales para que la aplicación servidor funcione. Se abrirá una consola de comando y se navegará hasta entrar en esta carpeta. Una vez allí se procederá a instalar las dependencias necesarias para que la aplicación servidor pueda funcionar. Para ello se ejecutará el siguiente comando:

- Npm install

Este comando instalará las dependencias que son necesarias para poder ejecutar la aplicación servidor.
Una vez instaladas las dependencias, es necesario escribir en la consola el siguiente comando:

- Npm run start

Por defecto este proyecto usa el puerto 3000 para entrar a la aplicación, por lo que al escribir en un navegador localhost:3000 se podrá visualizar la entrada de la aplicación.

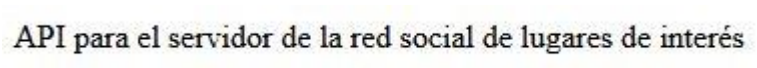


Figura 95: Visualización al iniciar la aplicación servidor en el navegador

Si se quiere entrar en modo de desarrollo tan sólo en necesario añadir al comando anterior la siguiente opción, *--dev.* (el punto no hay que incluirlo)

Apéndice B

Manual de Instalación del cliente

Requerimientos:

Para poder instalar tanto el cliente también es necesario tener instalado *NodeJS* y el gestor de paquetes *npm* (Node Package Management).

Como se han comentado los pasos para instalar estas herramientas, se procederá a la instalación de la aplicación cliente.

Instalación:

Para instalar la aplicación cliente es un proceso similar al de la aplicación servidor. En un primer momento se debe descomprimir la carpeta clienttfg que se adjunta en este proyecto. Se establecerá una carpeta con el mismo nombre. Se debe abrir la consola de comandos y navegar hasta entrar en esta carpeta.

Una vez en la carpeta a través de la consola de comandos se debe ejecutar *npm install*, al igual que con la parte del servidor, para poder intalar las dependencias esenciales para esta parte de la aplicación.

Para la aplicación cliente, además de estos paquetes, es necesario instalar el cli (command line interface) de Angular a través del siguiente comando:

- npm install -g @angular/cli

Al poner en el comando *-g*, conseguiremos que *Angular* esté disponible desde todo el sistema operativo ya que se instalará globalmente. Para verificar que *Angular* se ha instalado correctamente podremos ejecutar el comando *ng --version*, que indicará la versión instalada de *Angular*.

Para construir la aplicación cliente nos serviremos del siguiente comando:

- ng build

Con este comando Angular empezará a construir la aplicación cliente. Para arrancarla sólo es necesario utilizar el comando ng serve, que arrancará la aplicación.
Por defecto, se ha puesto que esta aplicación use el puerto 4200, por lo que al abrir el navegador y escribir localhost:4200 se podrá entrar al inicio de la aplicación.

Se ha configurado un proxy en esta aplicación cliente para evitar posibles problemas de Cors, por lo que si queremos entrar a la aplicación a través de proxy sólo deberemos escribir este comando por consola:

- ng serve --proxy-config.conf.json

Figura 97: Archivo con la configuración del proxy.

En la figura número 97 del apéndice se observa el archivo .json con el que se ha configurado el proxy. En principio todas las urls de la aplicación cliente apuntan al servidor donde se ha alojado este proyecto. Si se quiere modificar el proxy para que apunte al servertfg local, se debe modificar la url de target de https://servertfg.herokuapp.com/ a http://localhost:3000.

Lo mismo que con el proxy, si se desea que las peticiones que se hacen desde esta aplicación cliente vayan al servidor local en vez de al servidor que se ha subido en Heroku, es necesario modificar en los archivos service la misma url que en el archivo proxy.

Apéndice C

Manual de Instalación base de datos

Por defecto, la base de datos está subida a MongoDB, por lo que los datos almacenados que se recogen, muestran o editan son los subidos a la base de datos asociada al creador de este proyecto.

Si se desea instalar *MongoDB* personalizado para usar otra base de datos, es necesario abrirse una cuenta en https://www.mongodb.com/
Una vez iniciada la sesión se debe generar un nuevo clúster en *MongoDB Atlas*, y generar una nueva conexión.

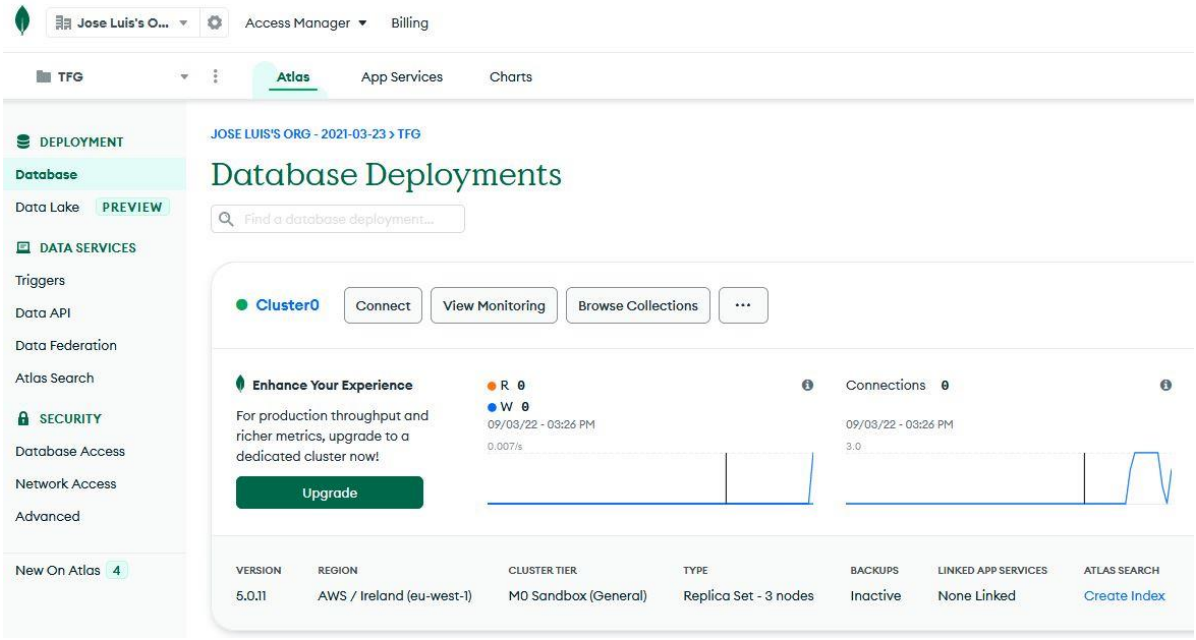


Figura 98: Clúster creado en MongoDB Atlas

Tras crear el clúster se debe generar una nueva conexión pulsando en *connect*.

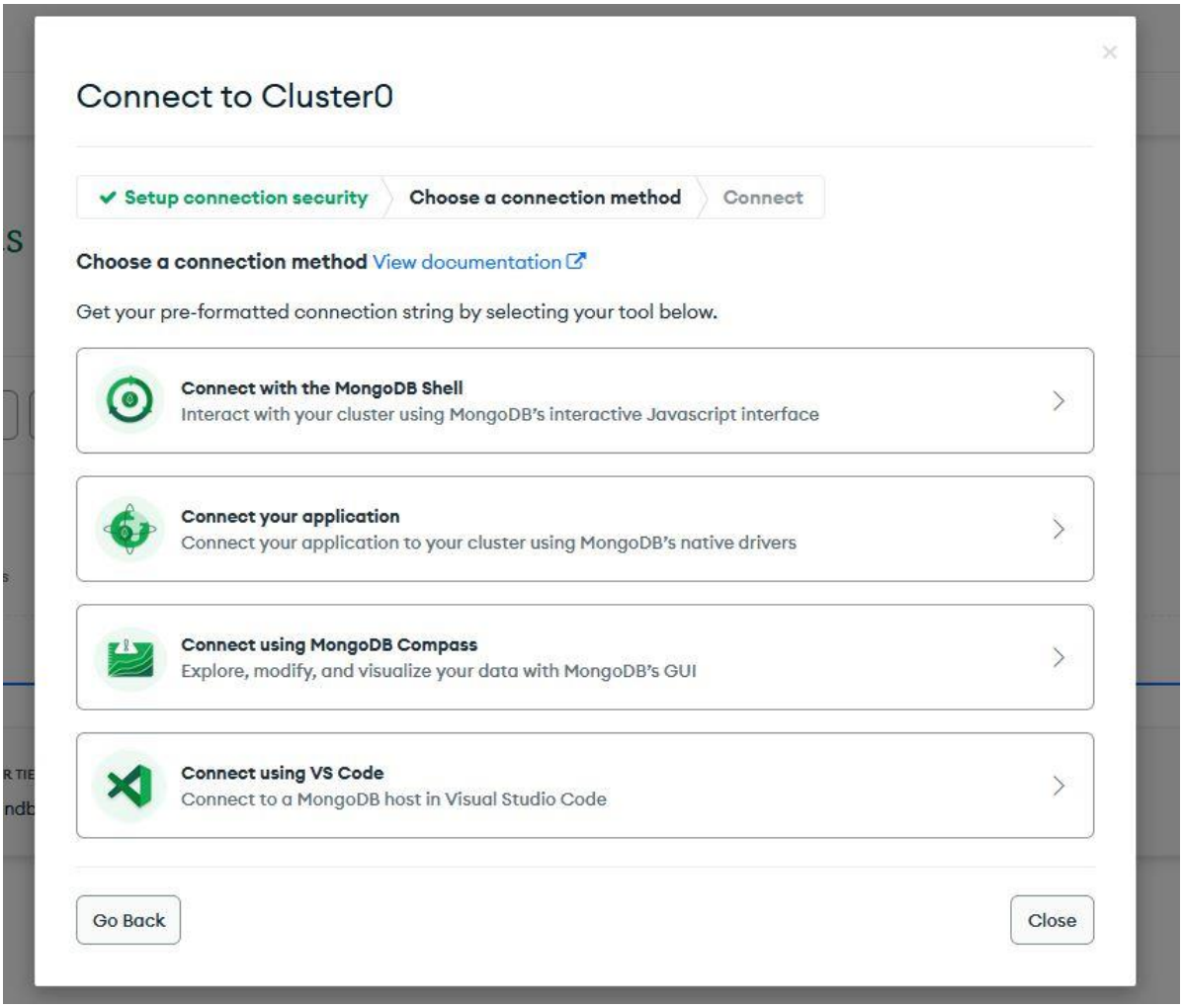


Figura 99: Conexiones disponibles de MongoDB

Al pulsar en *Connect your application* MongoDB nos dará la clave secreta y la URL para poder conectarnos a la base de datos.

En la figura 100 se muestra donde se deben introducir los parámetros del paso anterior. La clave secreta se pone en *KEY_SECRET*, mientras que la URL se debe poner en *MONGO_BBDD*.

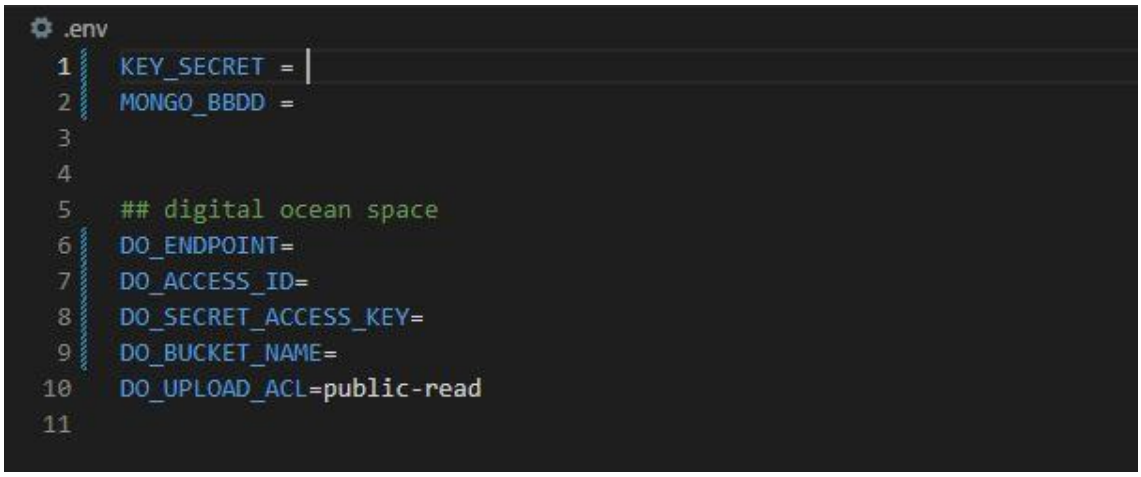


Figura 100: Archivo .env para la configuración

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga