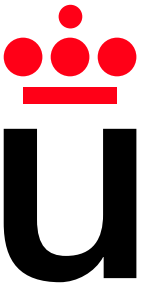

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería de la Ciberseguridad

Curso 2022-2023

Trabajo Fin de Grado

GESTIÓN AVANZADA DE LA IDENTIDAD EN FRAMEWORKS WEB

Autor: Álvaro Borreguero Nava
Tutor: Nicolás Rodríguez Uribe

Agradecimientos

Para empezar, quiero agradecer a todos los profesores que me han acompañado durante esta etapa en la universidad y fuera de ella, por haber fomentado el desarrollo de mi curiosidad, mis ganas de aprender y seguir siempre hacia delante y nunca tirar la toalla. También quiero agradecer a aquellos que me han formado como profesional, realizando un trabajo que nunca se podrá valorar lo suficiente.

No olvidarme de mi familia y personas que me rodean diariamente, que durante todo este periodo me han dedicado todo su tiempo, esfuerzo, apoyo, paciencia y dedicación con el fin de formarme y educarme de la mejor manera posible para afrontar la vida y todas las próximas etapas que comienzan a partir de ahora. Gracias de corazón.

Resumen

Debido al aumento de usuarios dentro de Internet cada vez son más los datos de los que las empresas disponen de sus usuarios y a la vez, son más las leyes que especifican cómo se deben tratar dichos datos. Es por esto por lo que surge este proyecto, con el que se trata de enfocar dicho problema desde el punto de vista de desarrollo.

El objetivo principal del proyecto ha sido el estudio e implementación de un buen tratamiento de datos de usuarios por parte de diferentes servicios. Es decir, saber gestionar estos datos de manera segura, realizar un control de acceso a nuestros servicios de manera personalizada y disponer de autenticaciones rápidas gracias a servicios externos a los nuestros.

El resultado ha sido la implementación de tres escenarios diferentes. En el primero, se desarrolla una página web con Spring Boot, donde existirán diferentes roles de usuario con diferentes permisos cada uno. Además, se incluirá una autenticación gracias a la tecnología OAuth2 que nos permite que los usuarios tengan una experiencia más fluida dentro de la web. Por otro lado, se desarrollarán dos servicios APIs diferentes, ambos desarrollados con Node aunque uno de ellos conectado a MySQL y otro conectado a Mongo. Estos servicios también dispondrán de control de acceso a través de usuarios registrados.

Palabras clave:

- Gestión de la Identidad
- Usuarios
- Seguridad
- Full Stack
- Node
- Spring Boot
- Bases de Datos
- API

Acrónimos

A continuación se definirán todos los acrónimos utilizados en este documento.

- SQL: Lenguaje de Consulta Estructurada.
- API: Interfaz de Programación de Aplicaciones.
- DB: Base de Datos.
- CSRF: Cross Site Request Forgery.
- SSRF: Server Side Request Forgery.
- JS: Java Script.
- TS: Type Script.
- HTML: Lenguaje de Etiquetas de Hipertexto.
- CSS: Lenguaje de Hojas de Estilo en Cascada.
- TFG: Trabajo Fin de Grado.

Índice de contenidos

Índice de figuras **XIII**

1. Introducción **1**
1.1. Contexto y alcance . . . 1
1.2. Estructura del documento . . . 2

2. Objetivos **3**
2.1. Problema . . . 3
2.2. Alternativas . . . 4
2.3. Metodología . . . 5
2.4. Vulnerabilidades . . . 6

3. Contenidos principales **8**
3.1. Docker y Docker Compose . . . 8
3.2. Bases de Datos . . . 9
3.2.1. MySQL . . . 9
3.2.2. MongoDB . . . 9
3.3. Visualización . . . 9
3.3.1. Dbeaver . . . 10
3.3.2. MongoDB Compass . . . 10
3.4. JavaScript . . . 10
3.5. TypeScript . . . 10
3.6. Node . . . 11
3.7. Spring Boot . . . 11
3.8. API REST . . . 11
3.9. Oauth2 . . . 12
3.10. Especificaciones . . . 12
3.10.1. Fase 1 . . . 13
3.10.2. Fase 2 . . . 13
3.10.3. Fase 3 . . . 14
3.11. Arquitectura . . . 14
3.11.1. Fase 1 . . . 14
3.11.2. Fase 2 . . . 14

3.11.3. Fase 3 . . . 15
3.12. Modelado de Datos . . . 15
3.12.1. MySQL . . . 16
3.12.2. MongoDB . . . 17
3.13. Implementación . . . 17
3.13.1. Spring Boot . . . 17
3.13.2. Node MySQL . . . 24
3.13.3. Node TS MongoDB . . . 27
3.14. Casos de Uso . . . 28
3.14.1. Usuario Público . . . 29
3.14.2. Usuario Autenticado . . . 29
3.14.3. Usuario Registrado . . . 29
3.14.4. Usuario Administrador . . . 30

4. Resultados **31**
4.1. Fase 1 - Web . . . 32
4.1.1. Usuario Público . . . 32
4.1.2. Usuario Autenticado . . . 32
4.1.3. Usuario Registrado . . . 33
4.1.4. Usuario Administrador . . . 34
4.2. Fase 2 - API JS - MySQL . . . 35
4.3. Fase 3 - API TS - MongoDB . . . 36

5. Conclusiones y trabajos futuros **44**
5.1. Conclusiones . . . 44
5.2. Trabajos Futuros . . . 45

Bibliografía **48**

Índice de figuras

2.1. Panel Notions 5

3.1. Flujo petición Oauth2 13
3.2. Arquitectura Fase 1 14
3.3. Arquitectura Fase 2 15
3.4. Arquitectura Fase 3 16
3.5. Modelo de Datos MySQL 16
3.6. Modelo de Datos MongoDB 17
3.7. Spring Boot - Resources 18
3.8. Spring Boot - Java 18
3.9. Spring Boot - Entry Class 19
3.10. Spring Boot - Controller 1 19
3.11. Spring Boot - Controller 2 19
3.12. Spring Boot - Mustache 20
3.13. Spring Boot - Services 20
3.14. Spring Boot - Compression 21
3.15. Spring Boot - Authentication 21
3.16. Spring Boot - OAuth2 Handler 22
3.17. Spring Boot - OAuth2 Config 23
3.18. Spring Boot - Security Configuration API 23
3.19. Spring Boot - Security Configuration Web 24
3.20. Node JS - Arquitectura Hexagonal 24
3.21. Node JS - Domain Comment Entity 25
3.22. Node JS - Infrastructure Repository 25
3.23. Node JS - Infrastructure MySQL Connection 25
3.24. Node JS - Infrastructure Routes 26
3.25. Node JS - Application UseCase 26
3.26. Node JS - Authentication 27
3.27. Node TS - Entity 28
3.28. Node TS - Class Entry 28
3.29. Node TS - Repository MongoDB 28
3.30. Node TS - Connection MongoDB 28
3.31. Caso de Uso Usuario Público 29

3.32. Caso de Uso Usuario Autenticado 29
3.33. Caso de Uso Usuario Registrado 30

4.1. Resultados - Home Page 32
4.2. Resultados - Login 33
4.3. Resultados - Register 34
4.4. Resultados - Login OAuth2 35
4.5. Resultados - Login Google 36
4.6. Resultados - Login Github 36
4.7. Resultados - Home Page 2 37
4.8. Resultados - Register 2 37
4.9. Resultados - Blog 38
4.10. Resultados - Post 38
4.11. Resultados - Comments 39
4.12. Resultados - New Post 39
4.13. Resultados - Panel 39
4.14. Resultados - Panel Admin 40
4.15. Resultados 2 - API Unauthorized 40
4.16. Resultados 2 - API Unauthorized 40
4.17. Resultados 2 - OpenAPI Docs 41
4.18. Resultados 2 - OpenAPI Docs Comment 41
4.19. Resultados 2 - Postman Comment Id 42
4.20. Resultados 2 - OpenAPI Docs 42
4.21. Resultados 2 - OpenAPI Comments 42
4.22. Resultados 2 - Postman Comments 43

1 Introducción

En este primer capitulo abarcaremos el contexto en el que nos encontramos antes de iniciar el proyecto y la estructura que sigue este documento para tener una comprensión del mismo.

1.1. Contexto y alcance

Este proyecto ha sido desarrollado con la idea principal de tener un mayor conocimiento sobre el uso de la identidad de los usuarios dentro de un servicio. Para ello se necesitaban una serie de escenarios diferentes para poder interactuar con los datos de los usuarios, incluso utilizando proveedores externos.

El proyecto consiste en tres escenarios diferentes, aunque relacionados entre sí, de manera que sea posible abarcar un mayor abanico de posibilidades. Estos escenarios, además, son entornos reales y comúnmente utilizados en el ámbito de desarrollo de software actual.

Por último, destacar que el "topic"de estos escenarios ha sido elegido de forma aleatoria, lo que significa que pueden ser adaptados según las necesidades de cada entorno.

1.2. Estructura del documento

A continuación se detallará la estructura utilizada en este documento para facilitar su lectura y comprensión.

En el capítulo 2 se explicarán los objetivos específicos de este proyecto, que irá acompañado de un estudio de diferentes alternativas para afrontarlo, además de un resumen de la metodología empleada.

En el capítulo 3 se detallará más en profundidad todo lo relacionado con el desarrollo del producto. Aquí se analizarán aspectos como las tecnologías utilizadas, las especificaciones del producto, así como su arquitectura y el modelado de datos que se ha utilizado. Por último, se tratarán aspectos más técnicos sobre la implementación y los casos de uso que tendrán los diferentes usuarios.

En el capítulo 4 podremos analizar los resultados obtenidos, que se dividen en las 3 fases desarrolladas en el capítulo anterior. Se explicará como interactúa el usuario con nuestros servicios, además de explicar las diferentes alternativas a su disposición dentro de nuestra aplicación.

Finalmente, en el capítulo 5 se detallarán las conclusiones obtenidas, además de futuras implementaciones o posibles alternativas a nuestra solución.

2

Objetivos

En este capítulo se detallarán los objetivos generales de este proyecto. Además, se analizan diferentes alternativas para su resolución. Por otro lado, explicaremos la metodología utilizada en el desarrollo para comprender como ha sido el flujo de trabajo.

2.1. Problema

La evolución de las páginas web ha tenido un crecimiento exponencial, principalmente por el aumento de usuarios que consumen internet diariamente. En los inicios una página web consistía en una plantilla HTML estática donde los usuarios consumían la información que se encontraba publicada. Con el paso del tiempo estas plantillas fueron ganando dinamismo, por ejemplo gracias a los estilos CSS. Después de esto, se empezó a tratar al usuario como la fuente de valor de estas páginas, empezando a interactuar con ellos e incluso almacenar información sobre estos.

Hoy en día, todas las páginas web almacenan una gran cantidad de datos de sus usuarios. Esto facilita la experiencia de uso gracias a procesos automatizados, como por ejemplo las autenticaciones o búsquedas anteriores, o por otro lado pueden ser utilizados para estudiar el comportamiento de nuestros usuarios dentro de la página web. Como vemos, el tratamiento de datos de usuario es algo bastante común y en que las empresas, cada vez más, deben invertir tiempo en su estudio y comprensión.

En este proyecto se va a tratar uno de estos aspectos comentados anteriormente, la gestión de la identidad de nuestros usuarios en nuestra página web. De esta forma comprenderemos cómo llevar a cabo esta autenticación en diferentes servicios, cómo almacenar toda esa información sensible de forma segura y por último cómo utilizar servicios externos para facilitar la autenticación de nuevos clientes en nuestra web. Todo el desarrollo tiene como objetivo su futura implementación en empresas o en diferentes proyectos.

2.2. Alternativas

Una vez analizado el problema se procedió al estudio de diferentes tecnologías para determinar cuáles eran las más convenientes. En este apartado señalaremos otras opciones que podían haber sido también utilizadas pero fueron finalmente descartadas.

En el apartado de almacenamiento de datos existen numerosos software diseñados para ello e incluso con almacenamiento en la nube como por ejemplo:

- **Amazon Aurora**: es un sistema de gestión de bases de datos relacionales. Está orientado al almacenamiento en la nube y dispone de numerosas ventajas tanto a nivel de seguridad como de costes. Por otro lado, se trata de una tecnología compleja de manejar.
- **IBM db2**: se trata de otro gestor de base datos con un gran potencial del que podemos destacar la capacidad de recuperación de datos, además de tener un gran soporte técnico y un alto rendimiento. El principal problema de esta tecnología es la limitación en la elección del hardware a utilizar.

Otro pilar fundamental es la implementación de nuestro servicio API Rest. Para ello podríamos haber seleccionado alguna de las siguientes tecnologías:

- **Golang**: se trata de un lenguaje de código abierto desarrollado por Google, de tipado estadístico y bastante polivalente. Este lenguaje no es tan maduro por lo que tiene un menor soporte por parte de la comunidad además de no ser tan eficiente con los recursos.
- **Flask**: se trata de un framework basado en Python, que ofrece numerosas extensiones para el mapeado y la carga de archivos. Esta librería no es conveniente para el desarrollo de grandes proyectos.

Para levantar todos los servicios de forma automática se podría haber utilizado la siguiente opción:

- **Kubernetes**: fue creado por Google utilizando el lenguaje Go y de código abierto. Se trata de una herramienta de gestión de contenedores que nos permite la creación y automatización de entornos de trabajo. En comparación con Docker, se trata de una herramienta más compleja de utilizar.

2.3. Metodología

Durante el desarrollo del proyecto se ha utilizado la metodología Kanban. Esta nos permite tener una relación directa entre el trabajo por hacer y la disponibilidad de que dispone el equipo. En este caso el equipo constaba de una única persona aunque la dinámica ha sido similar.

Para esto necesitamos diseñar un tablero con tres columnas principales (TODO, DOING y DONE) que nos permiten clasificar el trabajo en estos estados. Las tareas se dividen en tarjetas que se van moviendo en función del trabajo que se haya realizado. Como extra se ha incluido una puntuación a cada tarjeta para tener una mejor noción del esfuerzo necesario para cada una de ellas.

Destacar que esta metodología se puede adaptar a diferentes necesidades según cada proyecto. Podríamos introducir nuevas columnas como BLOCKED o incluso una TEST para indicar que una tarea está completada pero aún falta por testear.

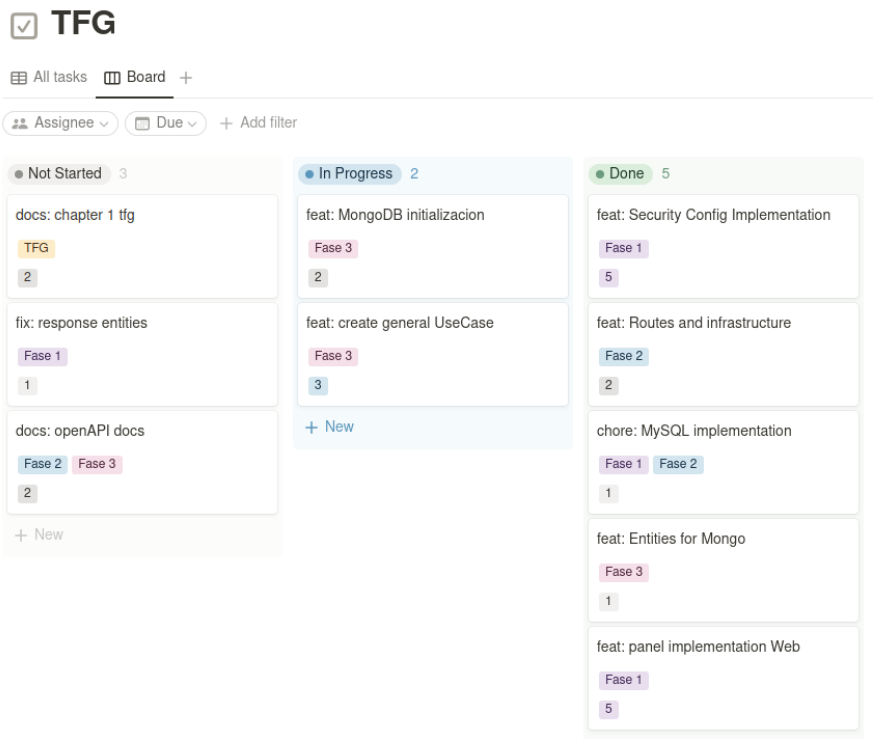

Figura 2.1: Panel Notions

Durante el desarrollo se ha utilizado la aplicación Notions para llevar a cabo todo este seguimiento. Como se observa en la figura disponemos de un tablero para la creación y actualización de todas las tareas.

2.4. Vulnerabilidades

En este apartado, se van a analizar algunas de las vulnerabilidades más comunes actualmente y, que por el tipo de implementación que se va a realizar, hemos tenido que tener en cuenta en todo momento.

- **Broken Access Control**: está enfocada en evitar que los usuarios se salgan fuera de los permisos de los que disponen. Es decir, evitar que un usuario realice acciones que no debería poder realizar. Para esto se necesita un buen control de acceso además de gestionar los permisos por roles. Para ello se utiliza el principio de mínimo privilegio.
- **CSRF**: esta técnica consiste en realizar peticiones entre sitios web diferentes por parte de la víctima, para forzarla a ejecutar código malicioso.
- **SSRF**: consiste en la falsificación de solicitudes en el lado del servidor, esto permite a un atacante inducir conductas dentro de los sistemas que le puedan beneficiar.
- **SQL Injection**: esta vulnerabilidad permite a los atacantes interactuar con las peticiones que se realizan a la base de datos. Modifican estas peticiones para poder acceder a información que no deberían ser visibles para ellos.

3

Contenidos principales

En este capítulo se analizarán las tecnologías utilizadas en el proyecto como Docker, las bases de datos de MySQL y MongoDB, diferentes herramientas de visualización y los lenguajes y frameworks utilizados.

Además, se explicará la arquitectura de cada una de las fases, así como el modelado de datos utilizado. Se finaliza explicando más en detalle la implementación y los resultados obtenidos, con casos de ejemplo.

3.1. Docker y Docker Compose

Docker es una herramienta que nos permite crear contenedores portables para que las aplicaciones software puedan ejecutarse en cualquier hardware o máquina, con el único requisito de tener Docker instalado. Tiene un modelo basado en imágenes, que permite compartir servicios con todas las dependencias predefinidas.

Por otro lado, Docker Compose permite ejecutar diferentes contenedores de manera sincronizada a través de un archivo YAML, haciendo posible crear arquitecturas de desarrollo mucho más complejas y dinámicas como veremos en la definición de la arquitectura en la Figura 3.4.

Se utiliza para levantar las bases de datos tanto MySQL como MongoDB, además de los servicios de Spring y Node. Para su implementación, es necesaria la utilización del archivo "docker-compose.yaml".

3.2. Bases de Datos

Una base de datos permite almacenar y conectar los datos de manera lógica para su futuro uso. Existen principalmente dos tipos de bases de datos dependiendo de cómo traten los datos:

- **Relacional:** tienen como base organizar la información en partes pequeñas a través de indicadores que permiten localizar los datos de forma directa. Utilizan un Lenguaje de Consultas Estructuradas (SQL).
- **NoSQL:** este tipo de bases de datos están enfocadas a datos específicos y disponen de esquemas más flexibles, por lo que suelen ser más fáciles de desarrollar.

3.2.1. MySQL

Se trata de una de las bases de datos utilizadas en el desarrollo, no la única pero sí la más popular, debido a su gran flexibilidad y a que se ha convertido en un estándar de la industria. Además, también dispone de un gran abanico de interfaces y aplicaciones desarrolladas en su entorno, como puede ser MySQL Workbench.

3.2.2. MongoDB

La segunda base de datos utilizada es MongoDB, en este caso es NoSQL y orientada a documentos, donde los datos se almacenan en objetos BSON. Una de sus características más importantes es la flexibilidad que aporta, debido a que objetos de las mismas colecciones pueden tener esquemas diferentes.

Otras ventajas que ofrece serían su capacidad de indexación, su capacidad de escalado y de balanceo de carga o la versatilidad de sus consultas ad hoc, con las que podemos realizar todo tipo de búsquedas, por campos, con determinados rangos o expresiones regulares.

3.3. Visualización

Las herramientas de visualización permiten acceder a las bases de datos de forma externa y poder comprobar y verificar dichos datos. Además, también ofrecen una forma rápida de interactuar con los mismos.

3.3.1. Dbeaver

Se trata de una herramienta de administración de bases de datos, que permite conectarnos tanto a SQL como NoSQL. Utiliza controladores JDBC para interactuar con las bases de datos relacionales y controladores propietarios para las NoSQL. Nos ha permitido conectarnos a nuestra base de datos de MySQL y poder interactuar con ella de forma sencilla.

3.3.2. MongoDB Compass

Es una aplicacion nativa de Mongo, que permite explorar e interactuar de manera sencilla con los documentos de distintas colecciones que forman una base de datos MongoDB. Tiene una gran cantidad de funciones que nos ayudan durante el desarrollo, como la facilidad y versatilidad de sus búsquedas en las colecciones, la posibilidad de exportar e importar archivos e incluso otorga información de eficiencia y almacenamiento.

3.4. JavaScript

Es un lenguaje de programación interpretado que apareció en 1995. Está basado en objetos y prototipos, es imperativo, de tipado débil y dinámico. Junto a HTML y CSS forma parte de los lenguajes esenciales en el desarrollo web. Originalmente era utilizado por parte del cliente, aunque con el paso del tiempo su uso se ha extendido al lado del servidor.

Actualmente cuenta con numerosas librerías y frameworks que facilitan ciertas tareas, así como el respaldo de una gran comunidad. Para el proyecto se está utilizando la ES2022.

3.5. TypeScript

Es un lenguaje construido en un nivel más alto que JavaScript, lo que quiere decir que extiende su sintaxis, pero aportando nuevas características, que permiten un desarrollo más sencillo y sólido. Algunas de estas características podrían ser el tipado fuerte, las anotaciones o módulos. Fue publicado por Microsoft en 2012 y dispone de un compilador escrito en TypeScript.

3.6. Node

Consiste en un entorno de tiempo de ejecución de JavaScript y de código abierto. Fue creado por los propios desarrolladores de JavaScript y publicado en el 2009. Node utiliza un modelo de entrada y salida sin bloqueo, en otras palabras, solicitudes y respuestas. Esto hace que sea bastante ligero y eficiente en aplicaciones en tiempo real. Actualmente sigue en desarrollo y a fecha de hoy se encuentran en la versión 20.3.1. aunque para el proyecto se han utilizado las versiones 16.20.1 y la 18.16.1.

Algunas de las principales ventajas de Node serían la facilidad de su aprendizaje, su capacidad para el escalado en aplicaciones, un buen manejo de peticiones simultaneas y la documentación disponible a través de su comunidad.

3.7. Spring Boot

Es un framework desarrollado para trabajar con Java y Spring, originado en 2002. Se trata de un entorno para desarrollo back-end de código abierto y gratuito. Tiene numerosas ventajas que han conseguido que sea un framework bastante utilizado por la comunidad.

Spring Boot ayuda a simplificar dependencias y configuraciones, automatizándolas y evitando al máximo los errores humanos en el desarrollo. Cada módulo funciona de manera independiente y aporta flexibilidad para desplegar los servicios de forma rápida y sencilla.

3.8. API REST

Es una interfaz de comunicación entre sistemas de información que permite comunicarnos con un sistema para obtener datos o ejecutar funciones de manera que este sea capaz de ejecutar dicha solicitud y llevarla a cabo.

La comunicación se realiza a través de diferentes llamadas como pueden ser GET, POST, PUT, PATCH o DELETE. Además, estas llamadas tienen asociado un status en la respuesta como por ejemplo, el 200 cuando la consulta ha sido exitosa.

El objetivo de una API Rest es poner a disposición del usuario una serie de rutas especificas para que este pueda hacer las llamadas que considere necesario para sus fines. Estas rutas permiten a la máquina del servicio saber qué lógica aplicar a cada petición. Estas peticiones pueden ir acompañadas de parámetros tanto en la misma ruta, conocidos como "query parameters", o a través de los

parámetros de la petición, lo que permite crear rutas más flexibles para el usuario.

Por último las API Rest pueden ser públicas, en las que cualquier usuario puede realizar las peticiones; o privadas, en las que el usuario tiene que autenticarse para que el servicio resuelva la petición. En muchas ocasiones se realiza un uso mixto, en que los usuarios no registrados tienen acceso a unas rutas específicas o están limitadas sus llamadas en función del tiempo entre las mismas.

3.9. Oauth2

Open Authorization o autorización abierta, es un estándar que permite que un sitio web o aplicación conceda acceso a sus recursos a otras aplicaciones en nombre del usuario. Sustituyó a Oauth 1.0 en el 2012 y ahora está considerado como un estándar en la industria.

Es un protocolo de autorización y no de autenticación, lo que permite conceder acceso a datos nuestros. Funciona utilizando tokens de acceso, no existe un formato especifico, aunque se suelen utilizar los tokens JWT (JSON Web Token). Estos tokens pueden almacenar información dentro de los mismos y suelen tener fecha de caducidad por temas de seguridad.

El flujo de esta autorización consiste de varios pasos:

1. La aplicación cliente solicita una autorización para acceder a los datos del usuario.
2. El usuario autoriza esta solicitud y la aplicación recibe una autorización confirmada.
3. La aplicación solicita un token de acceso al servidor de autorización con sus claves y con el permiso que ha recibido en el paso anterior.
4. Si las claves del cliente y la autorización son válidas, se emite un token de acceso que se envía a la aplicación cliente.
5. Con el token de acceso en su poder, y mientras este sea válido, la aplicación cliente ya podrá consultar los recursos a los que se le haya permitido el acceso.

3.10. Especificaciones

El proyecto ha sido planteado de forma incremental en 3 fases, que plantean escenarios diferentes que podemos utilizar tanto de forma conjunta como inde-

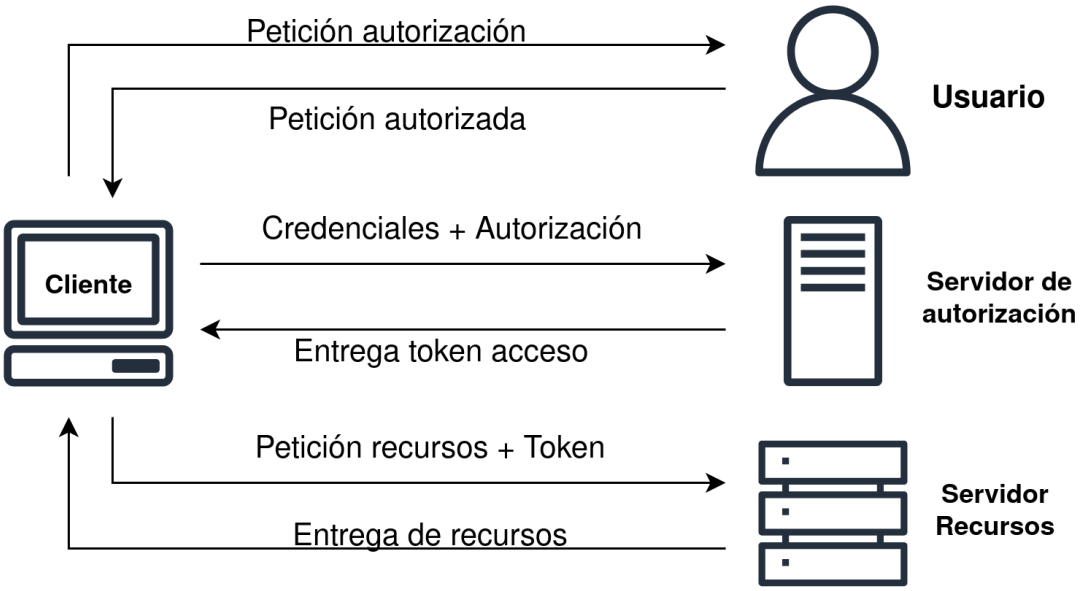

Figura 3.1: Flujo petición Oauth2

pendiente gracias a la implementación de contenedores Docker.

3.10.1. Fase 1

La primera fase del proyecto se basó en el desarrollo de un servicio web y API a través del framework Spring Boot, utilizando la base datos relacional MySQL. El servicio de API ha sido restringido a usuarios registrados, bloqueando así las peticiones a cualquier usuario sin registrar. Por otro lado la parte web tiene un sistema híbrido, existe contenido público y contenido privado al que sólo podrán acceder los usuarios de la web.

Además, también se ha implementado un sistema de autenticación con Google y con Github en la parte web. Para este uso se ha utilizado el protocolo Oauth2, lo que permite un mayor privilegio a los usuarios frente a los no registrados aunque siguen teniendo ciertas limitaciones de privilegios que solo eliminarán al completar el registro en la web.

3.10.2. Fase 2

La segunda fase del proyecto consistirá en realizar un servicio API independiente, pero que estuviera conectado a la misma base de datos que en la fase anterior. La API también estaba restringida a los usuarios ya existentes. Para esta fase se utilizó NodeJS y la base de datos ya montada de MySQL.

3.10.3. Fase 3

La última fase consistía en realizar un servicio API que estuviera conectado a otra base de datos diferente, en este caso una base de datos no relacional como es MongoDB. El servicio API fue desarrollado con Node utilizando el lenguaje de programación TypeScript y utilizando el mismo nivel de privilegios que en las fases anteriores.

3.11. Arquitectura

En este apartado se mostrarán diagramas lógicos de cada una de las fases para un mejor conocimiento de cómo interactúa cada uno de los servicios. Todo el proyecto ha sido realizado con Docker por lo que está configurado para desplegarse de forma automática gracias a sus contenedores.

3.11.1. Fase 1

Para la primera fase utilizamos dos contenedores. Uno de ellos es el propio de MySQL el cual se pone a disposición en el puerto 3306 y no está expuesto hacia el exterior, es decir, solo los propios contenedores tendrán acceso al mismo. Por otro lado, tenemos el servicio de SpringBoot que está configurado en el puerto 6868 tanto interna como externamente. Además podemos destacar dos rutas principales, una para el servicio web ("6868/") y otra para el servicio de la API ("6868/api").

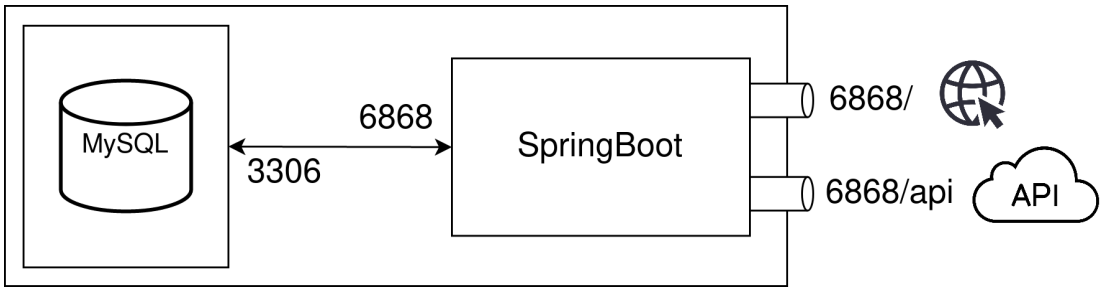

Figura 3.2: Arquitectura Fase 1

3.11.2. Fase 2

En esta segunda fase mantenemos la estructura inicial añadiendo un nuevo servicio. Esto lo hacemos con un nuevo contenedor con NodeJS el cual habilita

el puerto 3000 tanto interna como externamente. Para mostrar los datos desde esta API también se utiliza la conexión al puerto interno 3306 de la base de datos MySQL.

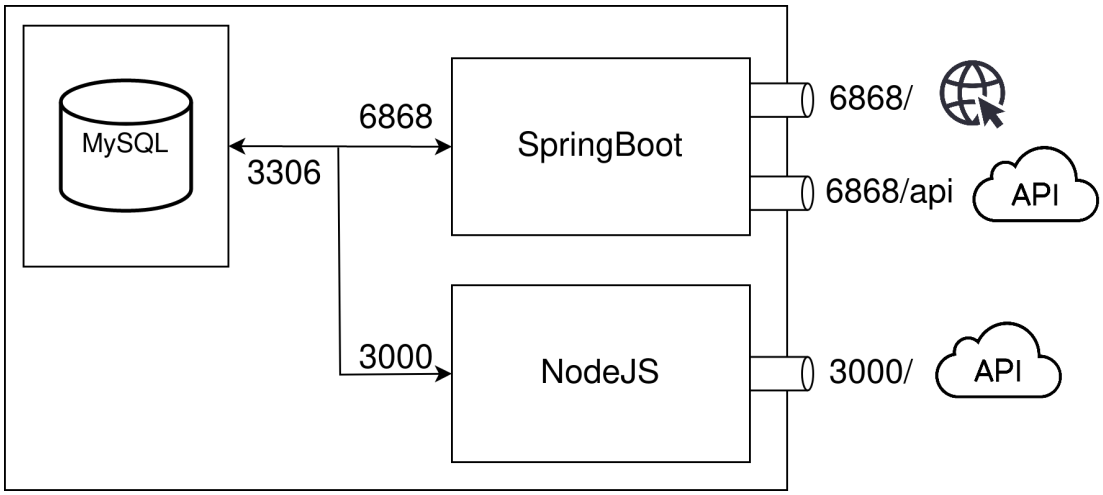

Figura 3.3: Arquitectura Fase 2

3.11.3. Fase 3

Para la última fase se incorporan dos servicios nuevos, uno para la base de datos de MongoDB y otro que también utiliza Node pero implementado con TypeScrypt. Como estándar con MongoDB se ha dispuesto el puerto 27017 internamente y el servicio con la API. En este caso dispone el puerto 3001 tanto interna como externamente.

3.12. Modelado de Datos

Debido a la existencia de dos bases de datos diferentes haremos una división entre los dos modelos de datos que existen y aunque ambos representan la misma información se almacena de forma distinta. Esto es debido a la forma de almacenar los datos dentro de una base de datos SQL y otra NoSQL. En el caso de MySQL las relaciones se generan de forma automática gracias a las PK Y FK y en el caso de MongoDB debemos crear nosotros esas relaciones entre ids numéricos.

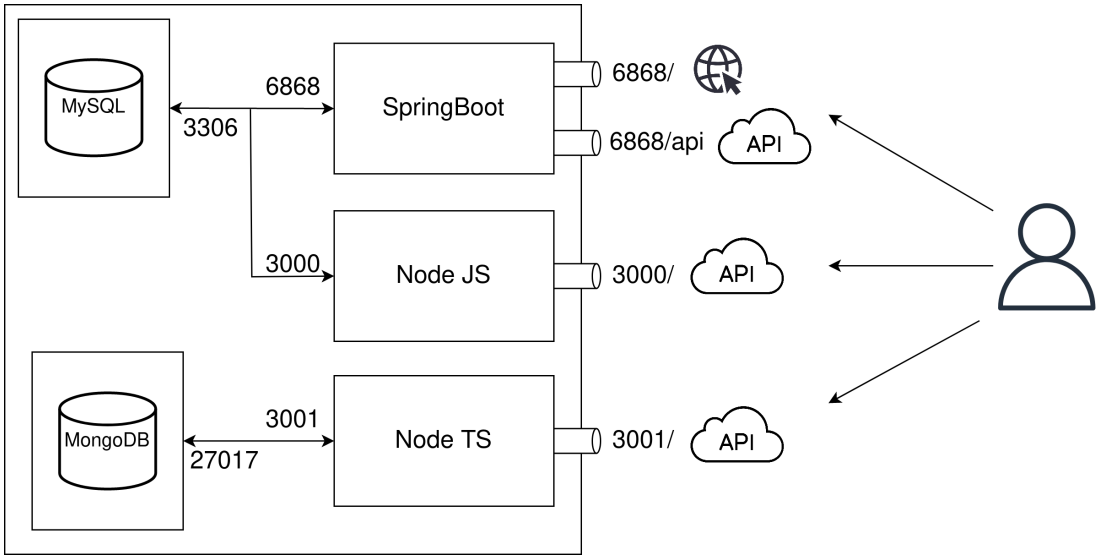

Figura 3.4: Arquitectura Fase 3

3.12.1. MySQL

Las tablas principales son ENTRIES, COMMENTS y USERS las cuales están relacionadas entre sí. Además, también podemos observar la tabla IMAGE perteneciente a la tabla de las entradas, y la tabla USER ROLES asociada a los usuarios.

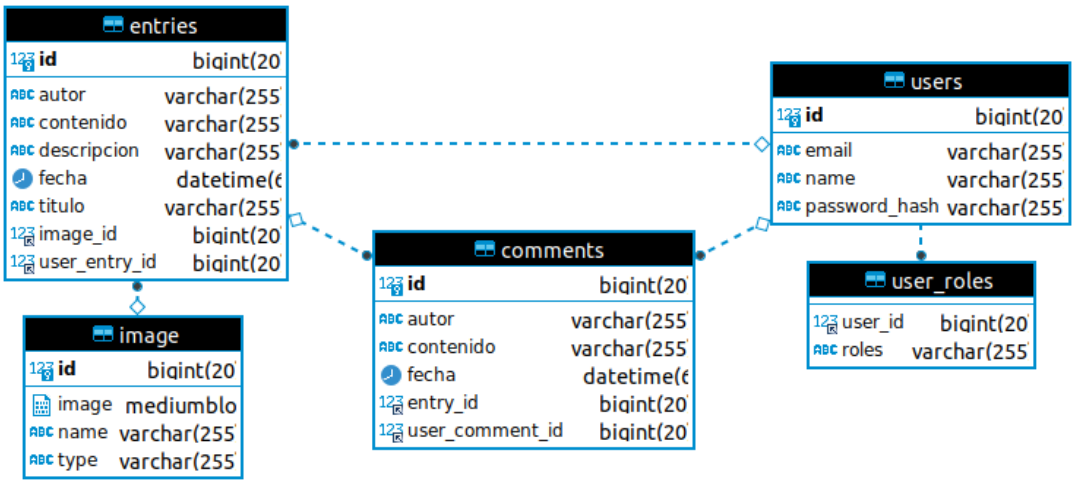

Figura 3.5: Modelo de Datos MySQL

3.12.2. MongoDB

En el caso de MongoDB se trata de una base NoSQL por lo que no existe esa correlación entre tablas que tenemos en MySQL. Pese a esto, se ha generado un modelado de datos que asemeje a una base de datos relacional y nos facilite la interacción con estos. Como vemos existen tres colecciones (USERS, ENTRIES y

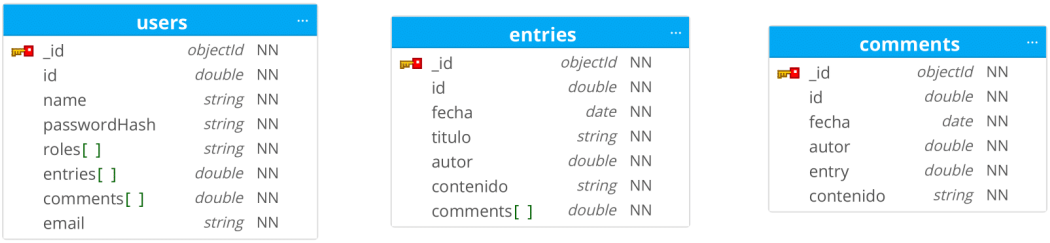

Figura 3.6: Modelo de Datos MongoDB

COMMENTS) aunque ahora no tienen esa relación natural de la base de datos. Si nos fijamos, podemos ver que los usuarios tienen un atributo ENTRIES y otro COMMENTS que almacenan el id de las entradas y los comentarios que tiene dicho usuario. En las entradas y los comentarios ocurre lo mismo. Esto permite que aunque no exista esa relación de tablas propia de una base de datos relacional, podamos crear esa relación lógica a nivel de código.

3.13. Implementación

A continuación se explicarán más en detalle los componentes y clases utilizadas, así como ciertas lógicas del código implementadas para el desarrollo de las diferentes fases del proyecto.

3.13.1. Spring Boot

La estructura de Spring Boot está formada por dos carpetas principales, una encargada de almacenar los elementos o plantillas que se van a utilizar y otra que almacena la lógica de implementación con los códigos en Java.

Como se puede ver, en la carpeta resources se almacenan las plantillas HTML que devolveremos al usuario y los elementos estáticos que serían las imágenes y las hojas de estilo para las plantillas en CSS. Lo único a destacar es la utilización de Mustache (como veremos en la Figura (3.12) en las plantillas HTML, que se explica más en detalle en el apartado relativo a los controladores de Java.

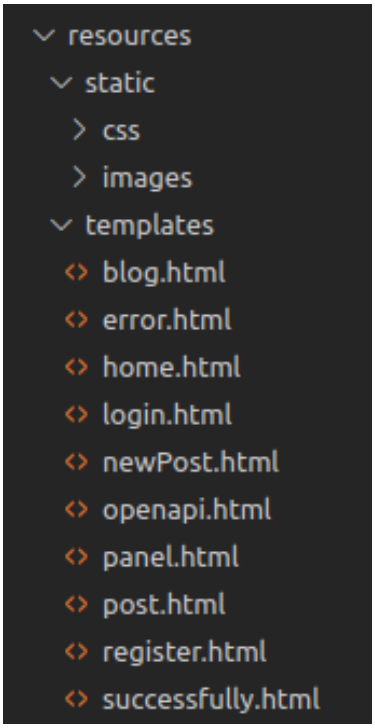

Figura 3.7: Spring Boot - Resources

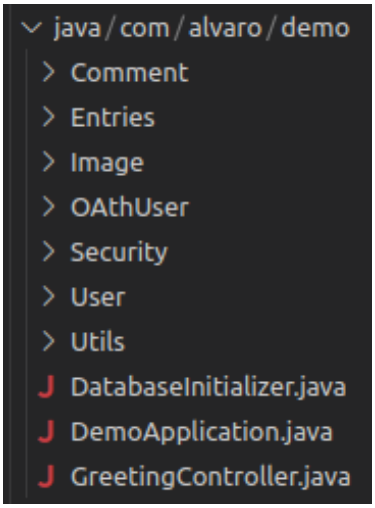

Figura 3.8: Spring Boot - Java

En la otra carpeta se almacenan subcarpetas asociadas con las clases y con la implementación lógica de la aplicación. En esta segunda parte se examina en profundidad ya que tiene una mayor complejidad.

- **Entidades:** Las entidades nos permiten definir las variables y métodos propios de un tipo de instancias para su posterior almacenado en la base de datos.

  Si nos fijamos en nuestro código tenemos las entidades de ENTRY, COMMENT, USER e IMAGE. Todas estas siguen la misma estructura, tienen una serie de atributos y van acompañadas de uno o más constructores. Además, los atributos pueden tener etiquetas asociadas, lo que indica comportamientos especiales para lo mismo:

  1. **Id**: indica el atributo identificador de la entidad, puede generarse automáticamente o podemos utilizar otros métodos.
  2. **CreationTimestamp**: establece la fecha en el momento que se crea la entidad, es propio de Hibernate.
  3. **JsonIgnore**: acompaña a los atributos que no deseemos que se muestren a los usuarios por la lógica de la aplicación o por seguridad, como las contraseñas.
  4. **Relaciones**: estas nos permiten indicar cuándo una entidad tiene como atributo otra entidad.

```java
@Entity
@Table(name="entries")
public class Entry {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    @CreationTimestamp
    private Date fecha;
    private String titulo;
    private String autor;
    private String contenido;
    @JsonIgnore
    private String descripcion;
    @JsonIgnore
    @OneToMany(cascade=CascadeType.ALL, mappedBy = "entry")
    private List<Comment> comments;
    @JsonIgnore
    @ManyToOne
    private User userEntry;
    @JsonIgnore
    @OneToOne(cascade=CascadeType.ALL)
    private Image image;
    public Entry(){}
    public Entry(Long id,String titulo, String contenido, User user, Image image){
```

Figura 3.9: Spring Boot - Entry Class

- OneToOne: tipo de correspondencia 1:1.
- OneToMany: tipo de correspondencia 1:M.
- ManyToOne: tipo de correspondencia M:1.

- **Controllers:** los controladores se encargan de recibir peticiones y aplicarles los procesos necesarios que se hayan definido para poder responder a dichas peticiones. Las principales anotaciones de estas funciones son: GetMapping, PostMapping, DeleteMapping, PutMapping y PatchMapping. Todas estas anotaciones vienen acompañadas de la ruta a la que van asociadas, permitiéndonos definir diferentes comportamientos por parte del servidor en función de la ruta o del tipo de petición que se haya realizado.

```java
@GetMapping("/")
public String home(Model model){
    return "home";
}
@GetMapping("/login")
public String login(Model model){
    return "login";
}
```

Figura 3.10: Spring Boot - Controller 1

```java
@GetMapping("/blog/{id}")
public String post(Model model, @PathVariable long id){
    Entry entry = entryService.getEntryById(id);
    if(entry != null){
        model.addAttribute(attributeName:"entry", entry);
        return "post";
    }else{
        model.addAttribute(attributeName:"property", HttpStatus.NOT_FOUND);
        return "error";
    }
}
```

Figura 3.11: Spring Boot - Controller 2

La última característica importante de los controladores es la capacidad que tienen para enviar variables a la parte del cliente, más concretamente a las plantillas HTML que vimos en la Figura 3.7. Gracias a Mustache, se puede interpretar dichas variables para interactuar con ellas en la página.

Como se puede observar, se tiene pleno control sobre estas variables, pudiendo hacer condicionales, recorrernos listas e incluso acceder a atributos de entidades predefinidos.

```
<div class="blog-container">
    {{#listEntries}}
    <div class="blog-box">
        <div class="blog-img">
            <img src='/blog/{{&id}}/image' alt="Img">
        </div>
        <div class="blog-text">
            <span>{{fecha}} / {{autor}}</span>
            <a href="/blog/{{&id}}" class="blog-title">{{titulo}}</a>
            <p>{{descripcion}}</p>
            <a href="/blog/{{&id}}">Read More</a>
        </div>
    </div>
    {{/listEntries}}
</div>
```

Figura 3.12: Spring Boot - Mustache

- **Services:** los servicios actúan como intermediarios entre los repositorios y los controladores, o cualquier parte del código que necesite comunicarse con la base de datos. Los repositorios disponen de funciones predefinidas y nuestra función en los servicios consiste en crear las lógicas necesarias para el correcto funcionamiento de nuestra aplicación, en lo que a la comunicación con la base de datos se refiere.

```
public Comment deleteComment(long id){
    if(commentRepository.existsById(id)){
        Comment toReturn = commentRepository.findById(id).get();
        toReturn.getEntry();
        commentRepository.deleteById(id);
        return toReturn;
    }else return null;
}
```

Figura 3.13: Spring Boot - Services

En la figura anterior se observa la función deleteComment en el servicio de los comentarios. Para definir dicha función se utiliza el repositorio de los comentarios, y se lleva a cabo las comprobaciones necesarias antes de eliminar el comentario. Además, se devuelve dicho comentario como respuesta.

- **Image Utils:** si nos fijamos en la entidad IMAGE, podemos apreciar que uno de sus atributos es un array de bytes donde almacenaremos dicha imagen en la base de datos. Para esto se han creado dos funciones que permiten a la hora de almacenar y leer dichas imágenes. Una de las funciones se encarga de comprimir la imagen para que ocupe menos espacio y así tener un mejor rendimiento y la otra se encarga de descomprimir la imagen para cuando deseemos mostrarla por la web. Para esta tarea utilizamos las clases DEFLATER y INFLATER propias de Java.

- **Users:** a continuación se explicarán dos implementaciones específicas sobre la entidad usuario que son esenciales para su correcto funcionamiento:

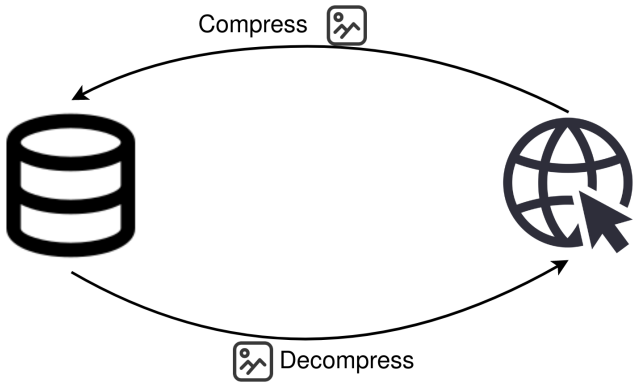

Figura 3.14: Spring Boot - Compression

- SessionScope: para llevar una correcta interacción con los usuarios es necesario implementar este componente. Permite mantener usuarios logueados e interactuar con ellos. Algunas de sus posibilidades serían conocer el usuario actual, saber qué permisos tiene o establecer un nuevo usuario logueado.
- Auth Provider: esta clase en concreto permite definir cómo autenticar a los usuarios dentro de nuestra aplicación, y será importante para futuros pasos.

```
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    String username = authentication.getName();
    String password = (String) authentication.getCredentials();
    User user = userRepository.findByName(username);
    if (user == null) {
        throw new BadCredentialsException(msg:"User not found");
    }
    if (!new BCryptPasswordEncoder().matches(password, user.getPasswordHash())) {
        throw new BadCredentialsException(msg:"Wrong password");
    } else {
        userComponent.setLoggedUser(user);
        List<GrantedAuthority> roles = new ArrayList<>();
        for (String role : user.getRoles()) {
            roles.add(new SimpleGrantedAuthority(role.toString()));
        }
        return new UsernamePasswordAuthenticationToken(username, password, roles);
    }
}
```

Figura 3.15: Spring Boot - Authentication

Como se observa existe una función encargada de llevar a cabo esta autenticación del usuario. Para ello utilizamos tanto el repositorio como el "SESSION SCOPEçitado anteriormente.

En este caso, estamos utilizando la librería de BCrypt para llevar a cabo la comprobación con el hash almacenado en la base de datos, correspondiente al usuario que se está autenticando. Además, gracias al "scope", y una vez comprobadas las credenciales, se establece la sesión con dicho usuario.

- **CSRF Handler:** esta clase lleva a cabo una implementación estándar, que permite implementar la utilización de tokens CSRF en nuestros formularios. De esta manera, se puede evitar un importante agujero de seguridad asociado al Cross Site Request Forgery.

  Estos tokens son enviados a las plantillas HTML para su utilización en todos los formularios. De esta forma, podemos verificar si una petición procede de nuestra aplicación o de una externa, que no poseerá dicho token y se previenen posibles ataques.

- **Oauth Handler:** esta clase se encarga de definir el comportamiento de nuestra aplicación cuando el usuario accede a través de una aplicación externa como Google o Github.

```
//GOOGLE
if(url.contains("google")){
    User user = userService.getByName(authenticationToken.getName());
    if (user == null) {
        name = authenticationToken.getPrincipal().getAttributes().get("given_name").toString();
        String email = authenticationToken.getPrincipal().getAttributes().get("email").toString();
        String[] roles = {"ROLE_USER_OAUTH"};
        user = new User(name,password:"",email,roles);
    }
    userComponent.setLoggedUser(user);
}
//GITHUB
if(url.contains("github")){
    String loginName = authenticationToken.getPrincipal().getAttributes().get("login").toString();
    User user = userService.getByName(loginName);
    if(user == null){
        Object email = authenticationToken.getPrincipal().getAttributes().get("email");
        if(email!=null){ email = email.toString(); }
        else { email = "";}
        String[] roles = {"ROLE_USER_OAUTH"};
        user = new User(loginName,password:"",email:"",roles);
    }
    userComponent.setLoggedUser(user);
}

this.redirectStrategy.sendRedirect(request, response,name!=null ? "/" :"/");
```

Figura 3.16: Spring Boot - OAuth2 Handler

Como se puede ver en la figura, tenemos la posibilidad de configurar el comportamiento dependiendo de la aplicación externa utilizada. En nuestro caso, solo disponemos de Google y Github, y en ambos se sigue el mismo estándar de loguear al usuario, aunque no se genera ese usuario en la base de datos. Por último, se define la ruta a la que será redirigido el usuario.

También destacar que será necesario definir nuestras credenciales para las páginas externas y así poder utilizar este servicio. Además, podemos especificar los datos a los que necesitamos acceder y así el usuario estará informado.

- **Security Configuration:** por último, seguramente uno de los archivos más relevantes, el Security Configuration. Este archivo permite configurar numerosos parámetros relacionados con la seguridad de nuestra aplicación. Lo primero a destacar, es que se encuentra dividido en dos configuraciones, una para el apartado web y otra para el servicio API. Ambas siguen una estructura similar, aunque con ciertos matices para que se haga posible el uso de ambos servicios de forma paralela y sin conflictos.

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            clientId: 68749933808-icnl71nsp6c9uqpkddvlsfpsafmf7tfq.apps.googleusercontent.com
            clientSecret: GOCSPX-[redacted]
            scope:
            - email
            - profile
          github:
            clientId: 5425eee9b52f31af2861
            clientSecret: [redacted]
            scope:
            - email
            - login
```

Figura 3.17: Spring Boot - OAuth2 Config

```java
@Bean
@Order(1)
SecurityFilterChain filterChainApi(HttpSecurity http) throws Exception{
    return http
            .securityMatcher(AntPathRequestMatcher.antMatcher(pattern:"/api/**"))
            .authorizeHttpRequests(request -> {
                request.anyRequest().authenticated();
            }).sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .httpBasic().and().csrf().disable()
            .build();
}
```

Figura 3.18: Spring Boot - Security Configuration API

Dento de la primera configuración, la encargada de gestionar el servicio de la API, podemos diferenciar varios apartados. El primero indica la ruta definida para esa configuración, que podemos ver que se indica como /api/**, es decir cualquier ruta que comience con la ruta /api. En segundo lugar, se establece que para realizar cualquier petición el usuario debe estar registrado a través de HTTPBASIC, con usuario y contraseña. Y por último, desactivamos el CSRF para que no interfiera en el servicio.

En nuestra segunda configuración se abarcan todas las rutas hacia nuestra aplicación, siempre y cuando no hayan pasado por la configuración anterior. En este caso existe una mayor diferenciación de permisos:

- **/panel /blog/post /blog/*** : Se necesita permiso de administrador o de usuario.
- **/blog** : Se necesita estar autenticado.
- **/**** : El resto de rutas estarán disponibles para cualquier usuario.

Después, tenemos una serie de configuraciones para la gestión de los inicios de sesión, donde destacaremos el uso del Auth Provider, ya definido en la Figura 3.15, para gestionar las credenciales. A continuación también podemos observar la configuración para el inicio de sesión a través de OAuth2 utilizando el Oauth2 Handler referido anteriormente y visto en la Figura 3.16. Para finalizar, activamos el CSRF para que sea utilizado en todas las rutas con excepción de la referente a la API.

```
@Bean
@Order(2)
SecurityFilterChain filterChainWeb(HttpSecurity http) throws Exception {
    return http
            .securityMatcher(AntPathRequestMatcher.antMatcher(pattern:"/**"))
            .authorizeHttpRequests(request -> {
                request.requestMatchers(...patterns:"/panel","/blog/post","/blog/*").hasAnyAuthority(...authorities:"ROLE_ADMIN","ROLE_USER")
                .requestMatchers(...patterns:"/blog").authenticated()
                .requestMatchers(...patterns:"/**").permitAll()
                .anyRequest().permitAll();
            })
            .authenticationProvider(authProvider)
            .formLogin(form -> form
                .loginPage(loginPage:"/login")
                .defaultSuccessUrl(defaultSuccessUrl:"/")
                .failureUrl(authenticationFailureUrl:"/error")
                .usernameParameter(usernameParameter:"username")
                .passwordParameter(passwordParameter:"password"))
            .sessionManagement(session -> session
                    .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
            .logout(logout -> logout
                .logoutUrl(logoutUrl:"/logout")
                .logoutSuccessUrl(logoutSuccessUrl:"/"))
            .oauth2Login(oauth -> oauth
                .loginPage(loginPage:"/login")
                .successHandler(oauth2authSuccessHandler))
            .csrf().ignoringRequestMatchers(...patterns:"/api/**")
            .and().build();
}
```

Figura 3.19: Spring Boot - Security Configuration Web

3.13.2. Node MySQL

En primer lugar hay que indicar que se ha trabajado siguiendo una arquitectura hexagonal, que permite una gran escalabilidad además de una gran adaptación a los cambios. Esta estructura basa su idea en tres capas, la de infraestructura, la de aplicación y la de dominio. Cada una de ellas tiene un objetivo propio aunque trabajan de forma conjunta para dar una buena respuesta a nuestra aplicación.

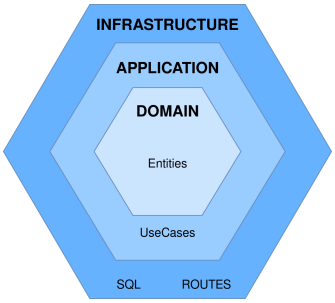

Figura 3.20: Node JS - Arquitectura Hexagonal

- **Domain**: Esta capa representa el núcleo de nuestra aplicación. En este caso es donde se han implementado todas las entidades necesarias para la correcta interpretación de los datos.

  Se definen aquellos atributos que va a poseer cada una de las clases involucradas en el desarrollo. Para esta implementación se han utilizado las entidades de COMMENT, ENTRY y USER.

```
class Comment{
    constructor(id, autor, contenido, fecha, entry_id, user_comment_id){
        this.id = id;
        this.autor = autor;
        this.contenido = contenido;
        this.fecha = fecha;
        this.entry_id = entry_id;
        this.user_comment_id = user_comment_id
    }
}
```

Figura 3.21: Node JS - Domain Comment Entity

- **Infrastructure**: La capa de la infraestructura se encarga de comunicarse con servicios externos o ser la parte visible de la aplicación. Dentro de esta capa podemos diferenciar dos apartados, el encargado de comunicarse con la base de datos y el que habilita las rutas para los usuarios.

  Para poder comunicarnos con la base de datos utilizamos los repositorios propios de cada entidad, que disponen de diferentes funciones como pueden ser getAll() - getById() para que sean utilizadas más adelante.

```
async function getAll(){
  const rows = await db.query(
    `SELECT *
    FROM comments`
  );
  const data = helper.emptyOrRows(rows);
  let listComments = [];
  for(let i in data){
    const newComment = new Comment(data[i]['id'],data[i]['autor'],
                                   data[i]['contenido'],data[i]['fecha'],
                                   data[i]['entry_id'],data[i]['user_comment_id']);
    listComments.push(newComment);
  }
  return listComments;
}
```

Figura 3.22: Node JS - Infrastructure Repository

Dentro de los repositorios, y para su correcto funcionamiento, se necesita otro archivo adicional para crear las conexiones con la base de datos. Recordar que para esta implementación se está utilizando MySQL, por lo que las sentencias vistas en la figura anterior utilizan su lenguaje nativo SQL.

```
const config = {
  db: {
      namedPlaceholders: true,
      host: process.env.MYSQLDB_HOST,
      port: process.env.MYSQLDB_DOCKER_PORT,
      user: process.env.MYSQLDB_USER,
      password: process.env.MYSQLDB_ROOT_PASSWORD,
      database: process.env.MYSQLDB_DATABASE,
  },
};

async function query(sql, params) {
  const connection = await mysql.createConnection(config.db);
  const [results, ] = await connection.execute(sql, params);

  return results;
}
```

Figura 3.23: Node JS - Infrastructure MySQL Connection

Dicha conexión a la base de datos dependerá en gran medida del lenguaje y el tipo de esta, aunque suele realizarse de forma sencilla a través de diferentes librerías.

El otro apartado importante de la capa de infraestructura corresponde al archivo ROUTES. Como su nombre indica se encarga de establecer las rutas de nuestro servicio API además de redirigir las peticiones según interese.

```
async function getCommentByIdHandler(req, res, next){
  try{
    const id = req.params.id ?? -1;
    res.json(await useCaseEntry.getById(id))
  }catch (err){
    console.error(`Error while getting entries `, err.message);
    next(err);
  }
}
function createCollectionsRouter(){
  let router = express.Router();

  router.get("/users",getAllUsersHandler);
  router.get("/users/:id",getUserByIdHandler);
  router.get("/entries",getAllEntriesHandler);
  router.get("/entries/:id",getEntryByIdHandler);
  router.get("/comments",getAllCommentsHandler);
  router.get("/comments/:id",getCommentByIdHandler);
```

Figura 3.24: Node JS - Infrastructure Routes

Como se puede observar, cada ruta disponible va asociada a una función encargada de manejar la petición del usuario. En estas funciones se utilizan los useCases, que se implementan en la capa de aplicación y se detallarán a continuación. De esta forma podemos abstraer la lógica de nuestra aplicación en la capa de infraestructura, la más externa, para que sea definida en capas más internas.

- **Application**: La última capa es la capa intermedia, llamada capa de aplicación. Esta se encarga principalmente de intermediar entre el núcleo y la parte externa de la estructura. Es aquí donde se implementan ciertas lógicas que como ya hemos señalado serán utilizadas por la capa de infraestructura. Estas lógicas están definidas en lo que se conoce como üseCase".

```
async function getAll(){
  const entries = await entryRepository.getAll();
  let listEntriesResponse = []
  for(let i in entries){
    const entryResponse = new EntryResponse(entries[i].id,entries[i].autor,
      entries[i].contenido,entries[i].descripcion,entries[i].fecha,entries[i].titulo);
    listEntriesResponse.push(entryResponse);
  }
  return listEntriesResponse;
}
```

Figura 3.25: Node JS - Application UseCase

Estas funciones dependerán mucho de la lógica que se desee implementar. En la figura anterior puede observarse una función sencilla utilizada para

obtener todos los "posts". Como se puede ver, se utilizan los repositorios para acceder a estos datos en MySQL, para después llevar a cabo un tratamiento de estos a través de las entidades de tipo Response", que las que recibe el usuario. Esto se realiza de esta manera para evitar filtrar información privada, como las contraseñas.

Para finalizar, podemos destacar en el useCase de los usuarios la función checkUser, utilizada por la APP para autenticar a los usuarios que realizan peticiones a nuestra API. Para implementar dicha función se prefirió ceder la lógica al repositorio específico.

```
async function checkUser(username,password){

  const rows = await db.query(
    `SELECT * FROM users WHERE name = :name`,{name: username}
  );
  const data = helper.emptyOrRows(rows);
  if(data.length>0){
    const hash = data[0]['password_hash'];
    if(await bcryptjs.compare(password,hash)){
      console.log("USER PETITION => " + username);
      return true;
    }else return false;
  }
  else return false;
```

Figura 3.26: Node JS - Authentication

Se trata de una comprobación sencilla gracias a la librería Bcrypt que permite comprobar si el hash de la contraseña enviada coincide con la almacenada para dicho usuario. Además, dentro de la sentencia SQL vemos como se envían las credenciales como variables, que permite evitar ataques de SQL Inyection.

3.13.3. Node TS MongoDB

La última fase tiene una arquitectura bastante similar, ya que se sigue utilizando la estructura hexagonal para el desarrollo. A continuación, se referirá a las partes especificas en el desarrollo de esta última fase, principalmente enfocadas en las conexiones con Mongo además de ciertas optimizaciones en el código.

- **Repositories**: Si recordamos en la fase anterior disponíamos de repositorios específicos para cada una de las entidades de la base de datos. Para evitar repetir código en el mismo proyecto, se ha implementado un repositorio genérico a todas las entidades.

  Para poder llevarlo a cabo, se han propuesto que todas las entidades procedan de la clase ENTITY, que a su vez se extendía de la clase DOCUMENT. Esto se puede implementar gracias a la librería de MongoDB.

```
export interface Entity extends Document {}
```

Figura 3.27: Node TS - Entity

```
export class Entry implements Entity{
    constructor(
        public id: number,
        public fecha: Date,
        public titulo: string,
        public autor: number,
        public contenido: string,
        public comments: number[],
    ){}
```

Figura 3.28: Node TS - Class Entry

Al tener las clases implementadas de esta forma podemos crear un repositorio único para la interfaz ENTITY. Como esta es implementada por cada una de las clases, y debido a la propia herencia, nos servirá para todas nuestras entidades.

```
export class MongodbRepository<E extends Entity> implements Repository<E> {
    constructor(
        private collection: Collection
    ) {}

    async readAll(): Promise<E[]> {
        const projection = { _id: 0 }; // eliminates _id field
        const allEntities = await this.collection.find().project<E>(projection).toArray();
        return allEntities;
    }
```

Figura 3.29: Node TS - Repository MongoDB

- **Conexión MongoDB**: La última diferencia principal es la conexión realizada a la base de datos. Gracias a las librerías que nos proporciona Mongo, esto se puede realizar en una única sentencia, facilitando aun más el manejo de base de datos.

```
export function createMongoClient(user: string, password: string, host: string, port: string): MongoClient{
    const connectionUrl = `mongodb://${user}:${password}@${host}:${port}`;
    return new MongoClient(connectionUrl);
}
```

Figura 3.30: Node TS - Connection MongoDB

Como se puede observar sólo es necesario conocer la dirección donde se encuentra alojada la base de datos, y en nuestro caso las credenciales para su acceso.

3.14. Casos de Uso

A continuación se especificarán las posibles interacciones de los usuarios dependiendo del nivel de privilegios que posean.

3.14.1. Usuario Público

En el caso del acceso de cualquier usuario público observamos que no dispondrán de muchas posibilidades. Únicamente podrá acceder a la pagina web para ver la página de bienvenida y poder registrarse.

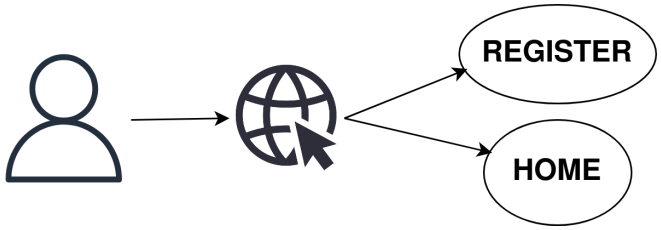

Figura 3.31: Caso de Uso Usuario Público

3.14.2. Usuario Autenticado

Un usuario autenticado a través de Google o de Github tendrá una mayor accesibilidad, ya que podrá acceder al blog genérico aunque no podrá acceder a ninguna publicación. Además, tendrá la posibilidad de autocompletar su registro.

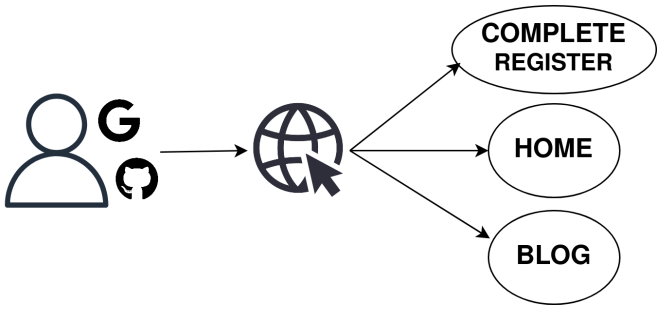

Figura 3.32: Caso de Uso Usuario Autenticado

3.14.3. Usuario Registrado

Por último, el usuario registrado tiene acceso a un gran abanico de posibilidades. Por la parte web tendrá acceso a todos los post, además de poder postear y comentar. También tendrá acceso a un panel personal en el que podrá gestionar su contenido y podrá deslogearse en cualquier momento. Por la parte de las APIs tendrá acceso los datos dispuestos tanto de MySQL como a los de MongoDB.

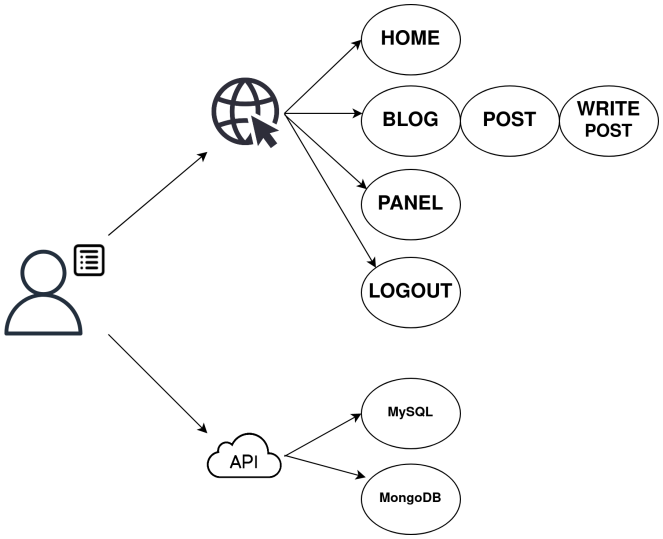

Figura 3.33: Caso de Uso Usuario Registrado

3.14.4. Usuario Administrador

Por último, recalcar que existe el rol de administrador, con el cual tenemos acceso total. Podríamos eliminar contenido de otros usuarios en la web e incluso crear nuevos usuarios.

4

Resultados

En esta sección se describen los resultados obtenidos en el TFG una vez finalizado el proyecto. En primer lugar, se analizará la Fase1 con los roles especificados en los casos de uso, y después se realizará una prueba frente a las APIs dispuestas en la Fase 2 y Fase 3 del desarrollo.

4.1. Fase 1 - Web

A continuación se especificarán los resultados diferenciados en función del usuario que se disponga, se trata de la fase Web por lo que son muy interactivos y llamativos para la experiencia de usuario.

4.1.1. Usuario Público

Aquel usuario que no realice ningún tipo de registro o autenticación tendrá el rol de Usuario Público y únicamente tendrá acceso a nuestra página principal donde será recibido y tendrá un mínimo de interacción.

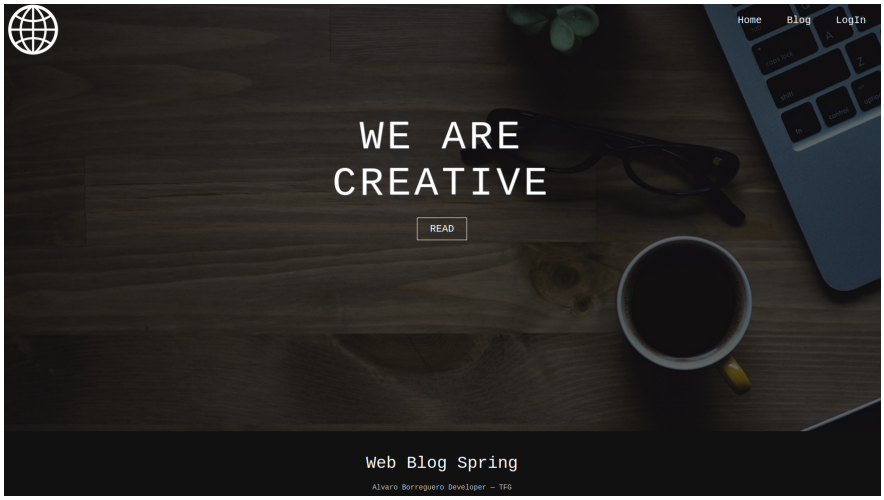

Figura 4.1: Resultados - Home Page

En el caso de que el usuario intente acceder a la página de Blog o al apartado de registro será enviado a la página para que realice el inicio de sesión.

También dispone de acceso a la página de registro para crear una cuenta en caso de que se trate de un nuevo usuario.

4.1.2. Usuario Autenticado

Un usuario autenticado es aquel que ha decidido iniciar sesión gracias a la autenticación de un tercero. Como se puede observar, la página dispone de autenticación con Google o con Github, el usuario es libre de elegir.

Cada icono redirige a la página de autenticación elegida, donde necesitará iniciar sesión para después ser redirigido de nuevo a nuestra web una vez autenticado.

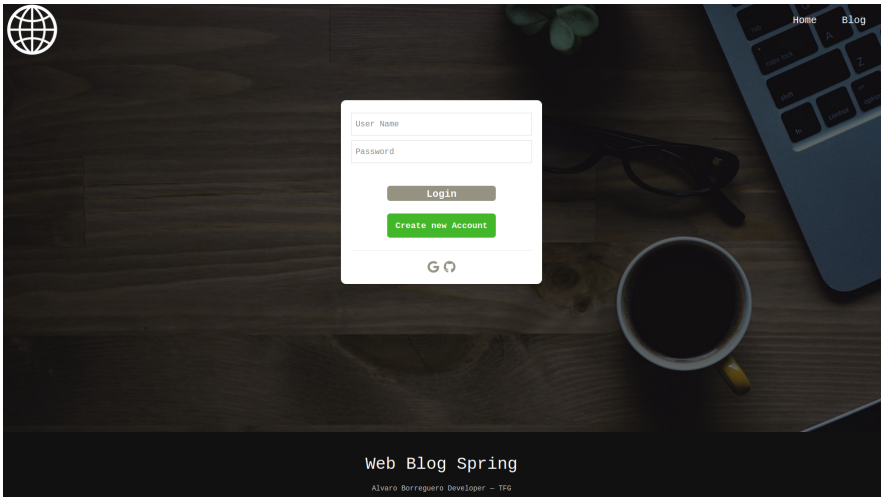

Figura 4.2: Resultados - Login

Una vez autenticado el usuario, podemos observar que la página principal ha cambiado y ahora dispone tanto del botón para deslogearse como un botón extra para completar el registro.

En el caso de que el usuario decida completar su registro, accederá a una pestaña donde se autocompletaría con su nombre (proporcionado por la página externa) para pasar a ser un usuario registrado.

Por último, como usuario autenticado tendrá acceso a la visión general de todos los post en la pestaña BLOG, donde verá un pequeño resumen de todos ellos, aunque aún no disponga de permisos para acceder individualmente a cada uno.

4.1.3. Usuario Registrado

Una vez disponga de un usuario registrado en la aplicación podrá realizar muchas más acciones debido a sus privilegios. La principal ventaja que dispone un usuario registrado es la posibilidad de ver todos los posts que existen en la página e incluso realizar comentarios en estos.

Además, si nos fijamos en la barra de control dispondrá de dos nuevas utilidades. Una de ellas permite crear nuevos post para la página, donde podrá elegir el título, la imagen y el contenido del mismo.

La segunda pestaña hace referencia al panel de control de cada usuario, en el que además de poder visualizar todos sus posts y comentarios, podrá eliminar cualquiera de estos elementos cuando desee gracias al icono de borrado.

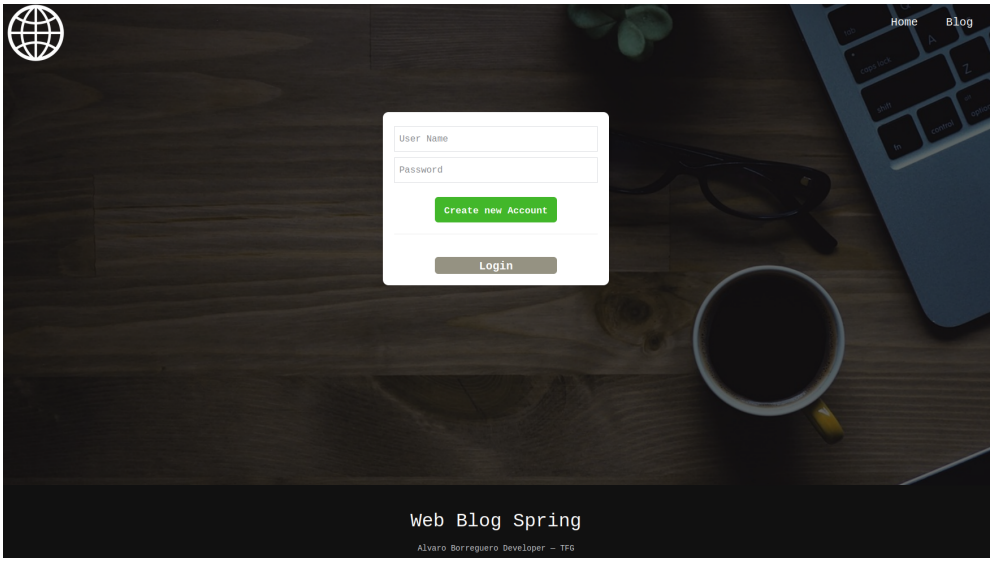

Figura 4.3: Resultados - Register

4.1.4. Usuario Administrador

El último de los roles que existen en la aplicación sería el de un usuario administrador. Este, además de tener todas las funcionalidades ya mencionadas, podrá gestionar todos los post y comentarios del resto de usuarios, teniendo pleno control en el blog.

Como se muestra a continuación, en el panel de control de un administrador se muestran todos los post independientemente de que hayan sido escritos por el administrador o no, al igual que los comentarios. De la misma forma que los usuarios normales, también dispone del icono de borrado para cada uno de estos elementos.

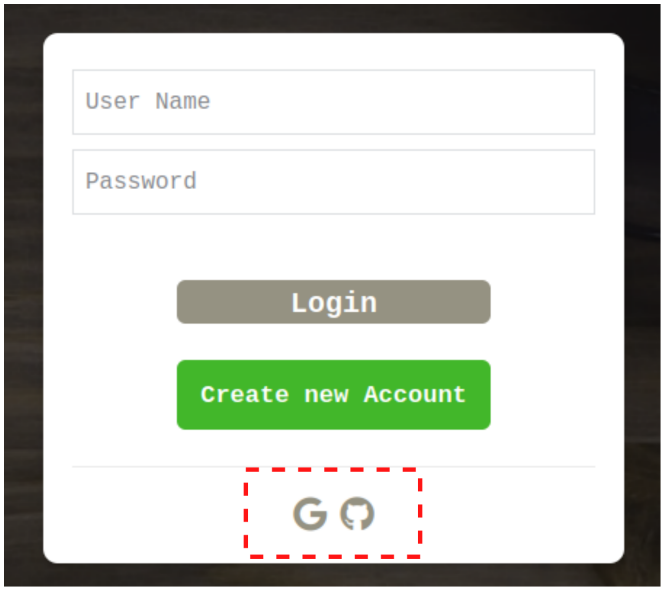

Figura 4.4: Resultados - Login OAuth2

4.2. Fase 2 - API JS - MySQL

En este segundo apartado se mostrará cómo se interactúa con nuestra API dispuesta a través de NodeJS conectada a la base de datos de MySQL. En primer lugar explicar que se pueden utilizar diversas aplicaciones que sirvan para este propósito. En este caso utilizaremos PostMan para realizar las peticiones.

Recordemos que las peticiones deben realizarse al puerto 3000 dispuesto en nuestra propia máquina, por lo que apuntaremos a la IP local. Al realizar una petición de prueba se puede ver que el servicio bloquea nuestra petición en caso de que no se utilice ninguna autenticación en dicha petición.

Para poder autenticarnos utilizaremos el apartado del propio software y estableceremos la autenticación con BASIC AUTH, donde se especificará un usuario válido en la aplicación.

Al utilizar el usuario administrador sí obtenemos respuesta por parte de la API, en este caso se ha obtenido una lista con los usuarios existentes dentro de la aplicación, filtrando datos sensibles como contraseñas.

Para que los usuarios dispongan de una guía se ha implementado una documentación gracias al estándar de openAPI, que permite indicar las diferentes peticiones que acepta nuestra API, así como sus parámetros, en el caso de que sean necesarios, o sus respuestas.

Como vemos en la ruta /comments/id se especifica que debemos introducir a través de la URL el id del comentario que deseemos. También se muestra el

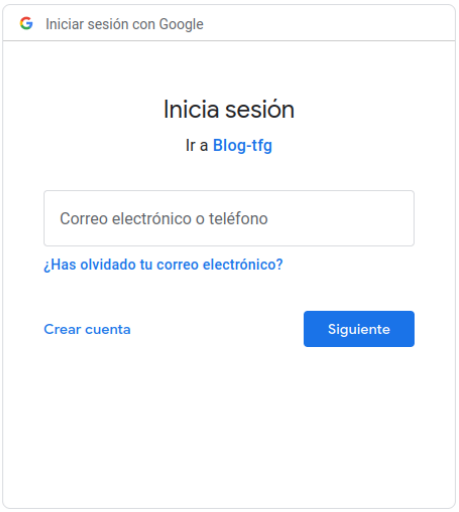

Figura 4.5: Resultados - Login Google

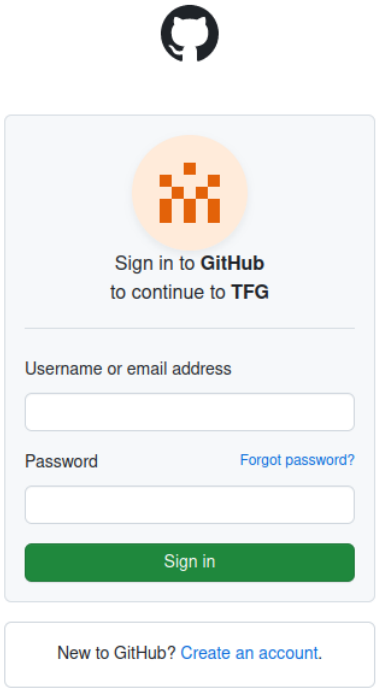

Figura 4.6: Resultados - Login Github

objeto que será devuelto indicando toda su estructura y el tipo de sus atributos.

Si realizamos la llamada a través de Postman podemos corroborar dicha información y ver como todos los campos de la respuesta coinciden con la documentación aportada.

4.3. Fase 3 - API TS - MongoDB

La experiencia de usuario en esta tercera fase es muy similar a la fase anterior. En primer lugar, también disponemos de documentación para que el usuario conozca las rutas disponibles dentro de nuestro servicio.

En este caso disponemos de 3 rutas para interactuar con la base de datos. Recordemos que este servicio se encuentra disponible en el puerto 3001 y que se sigue utilizando un control de acceso a través de usuarios registrados.

Podemos destacar que en este servicio se ha decidido utilizar parámetros a través de las queries como puede verse en la documentación. Estos son opcionales y permiten que el usuario realice peticiones más específicas según sus necesidades.

Para la ruta asociada a los comentarios existen 2 parámetros que serian el id

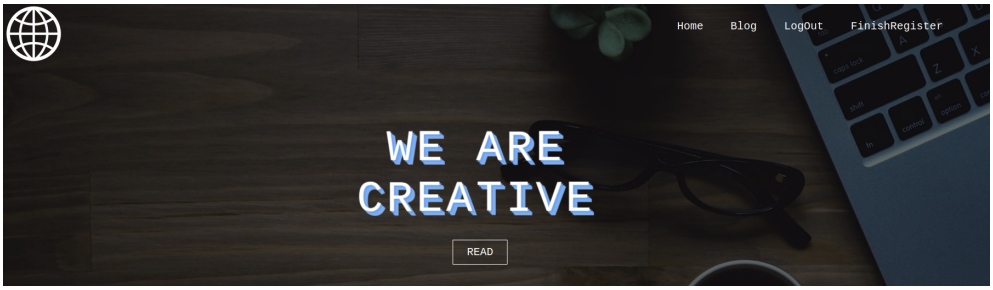

Figura 4.7: Resultados - Home Page 2

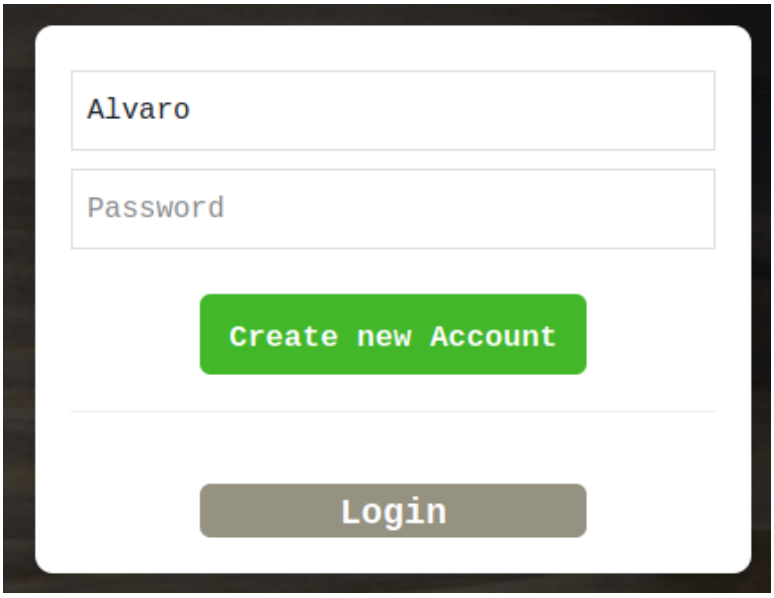

Figura 4.8: Resultados - Register 2

del autor que realizó el comentario o el id del post al que pertenece. Si realizamos esta petición con Postman se comprueba cómo la API nos devuelve el elemento que deseamos.

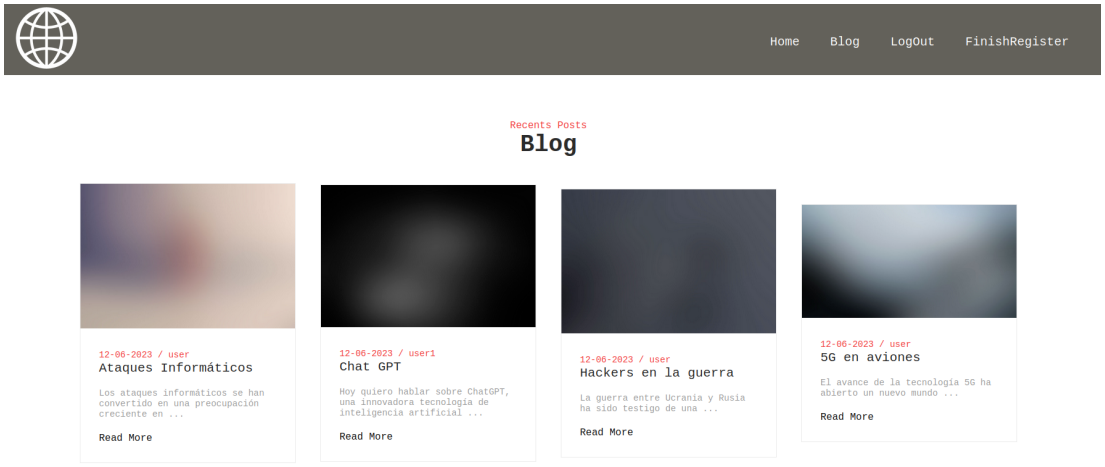

Figura 4.9: Resultados - Blog

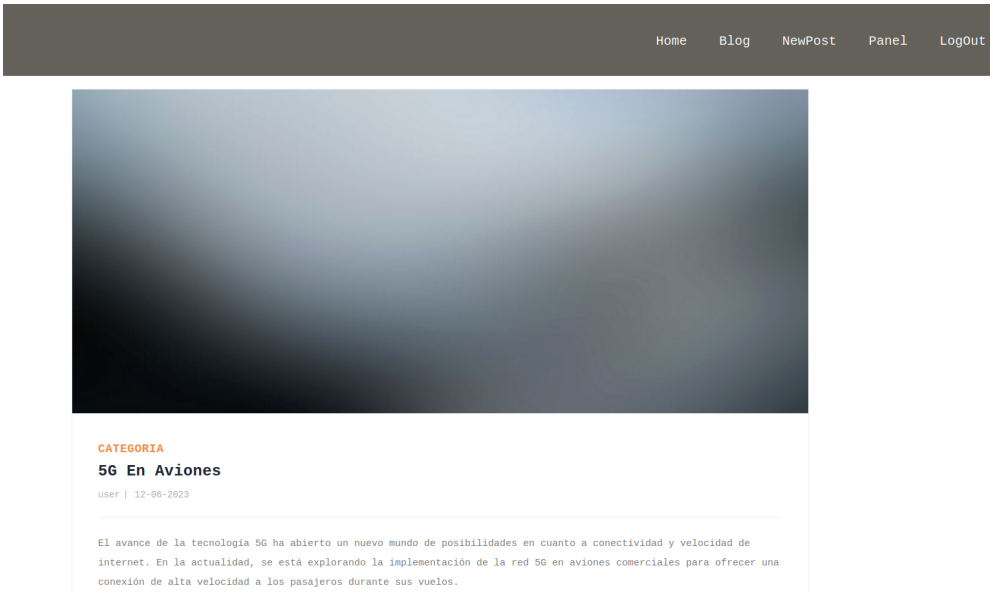

Figura 4.10: Resultados - Post

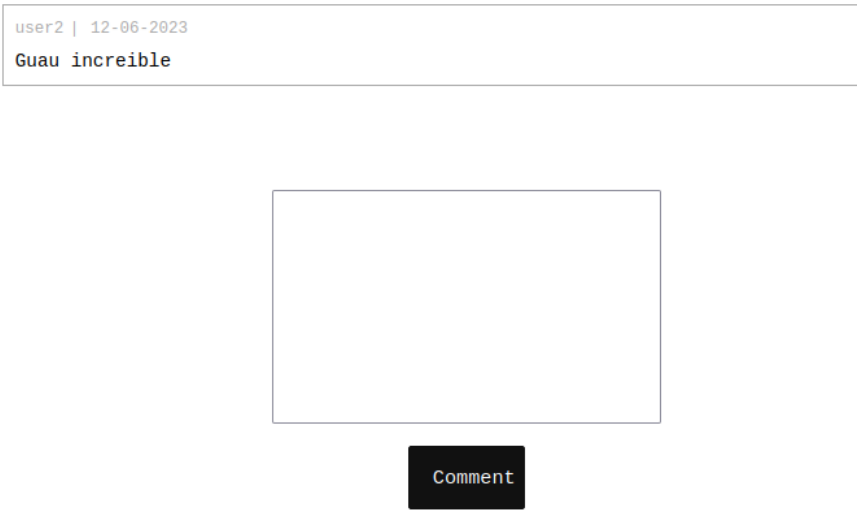

Figura 4.11: Resultados - Comments

Figura 4.12: Resultados - New Post

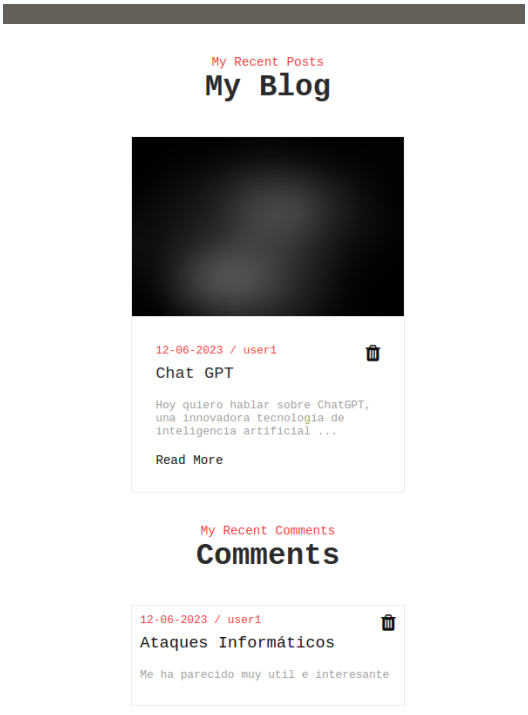

Figura 4.13: Resultados - Panel

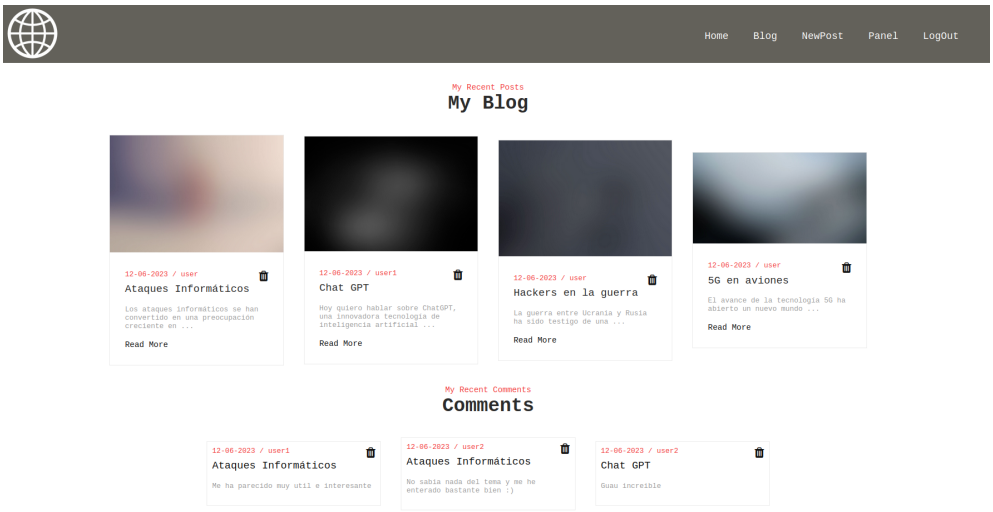

Figura 4.14: Resultados - Panel Admin

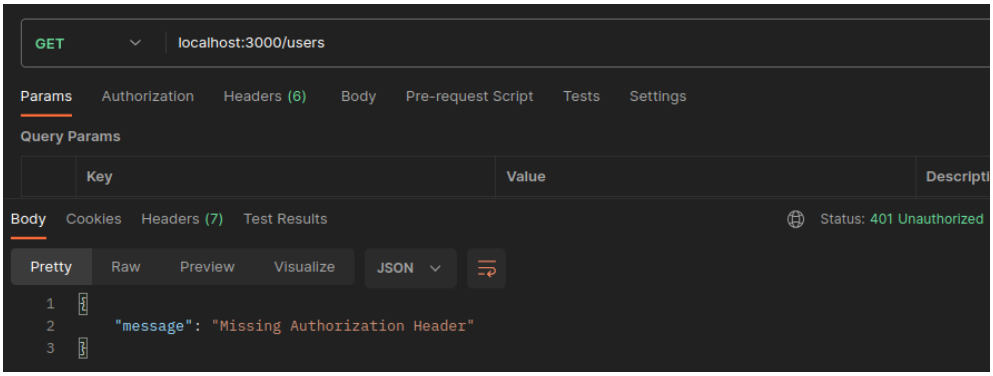

Figura 4.15: Resultados 2 - API Unauthorized

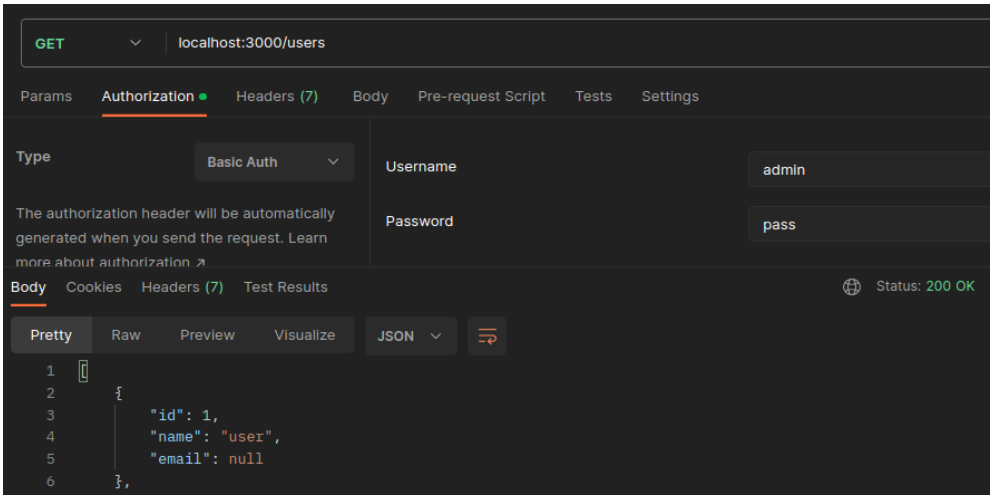

Figura 4.16: Resultados 2 - API Unauthorized

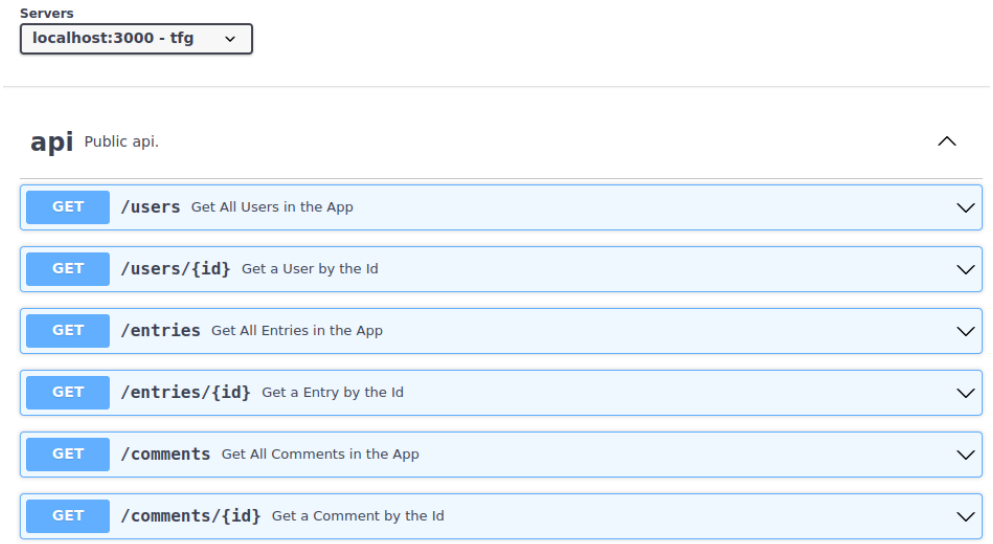

Figura 4.17: Resultados 2 - OpenAPI Docs

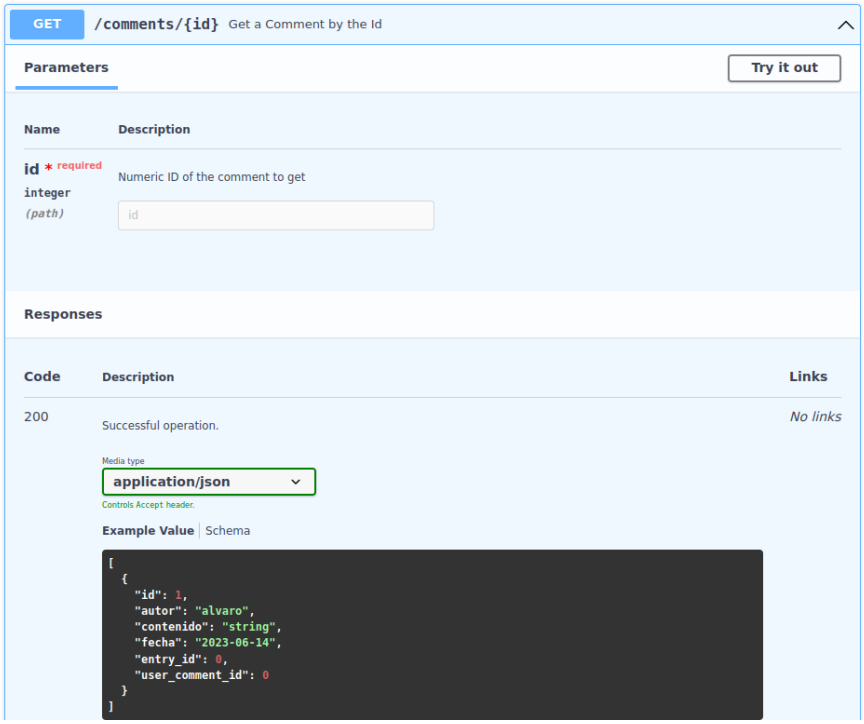

Figura 4.18: Resultados 2 - OpenAPI Docs Comment

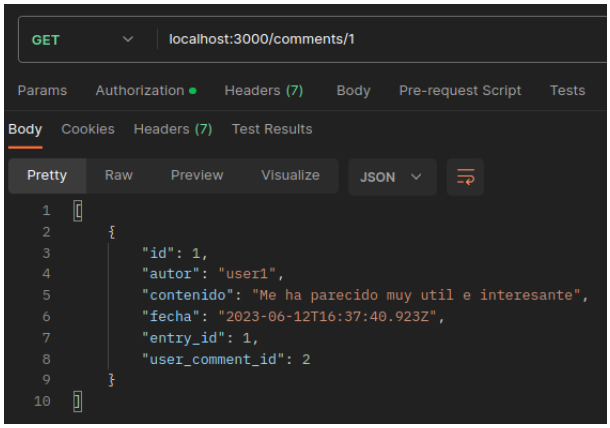

Figura 4.19: Resultados 2 - Postman Comment Id

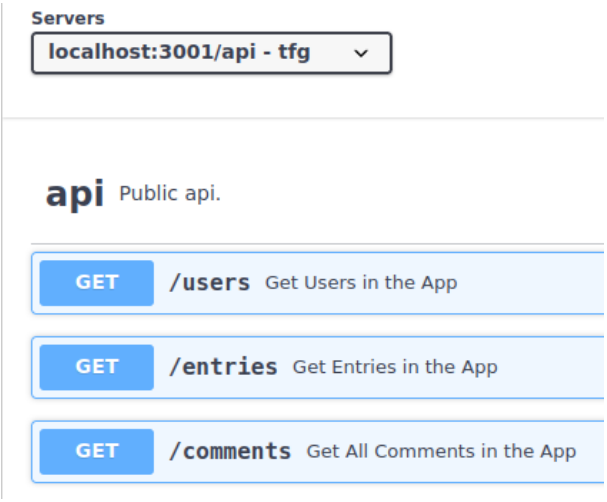

Figura 4.20: Resultados 2 - OpenAPI Docs

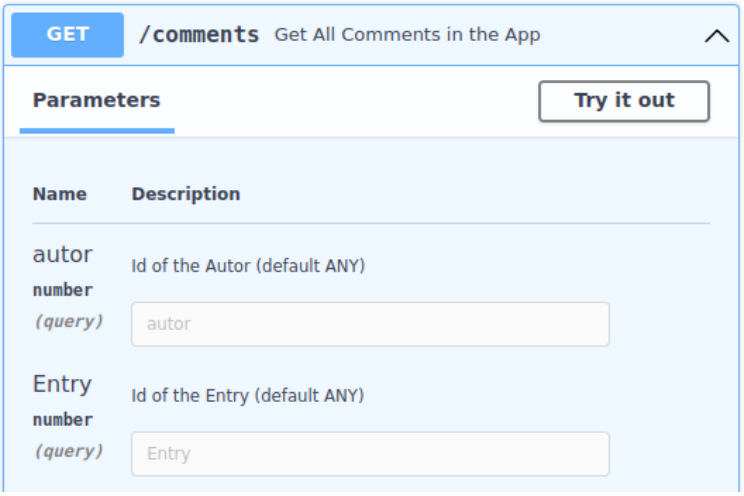

Figura 4.21: Resultados 2 - OpenAPI Comments

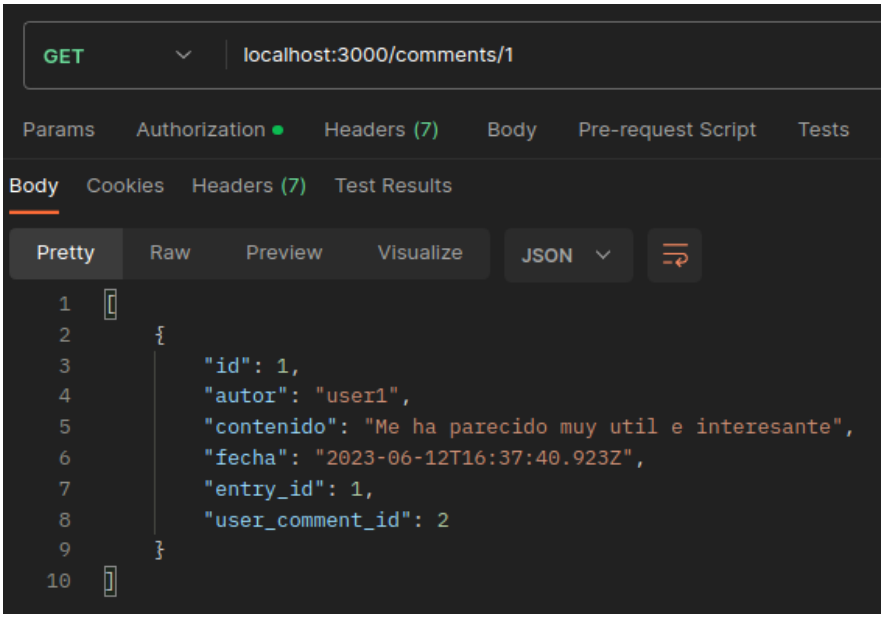

Figura 4.22: Resultados 2 - Postman Comments

5

Conclusiones y trabajos futuros

El objetivo de este capítulo es analizar las conclusiones obtenidas una vez finalizado el proyecto, destacando también los problemas que han surgido y posibles futuras implementaciones que ayudarían a mejorar o continuar con el mismo.

5.1. Conclusiones

Este proyecto surge como una manera de investigar y probar nuevos conceptos en la gestión de usuarios. Además, otra de las primicias consistía en realizar un desarrollo organizado y escalable. Con todo esto, se ha conseguido una base sólida de la que partir para futuras implementaciones con gran capacidad de escalado y de adaptación debido a la neutralidad del tema seleccionado, en este caso un blog.

A continuación de detallarán los objetivos que se han cumplido una vez finalizado el desarrollo del proyecto:

- Crear una aplicación web interactiva con el usuario, que sea dinámica y se pueda adaptar a las necesidades de diferentes clientes.
- Desplegar un sistema de API Rest donde se pongan a disposición los datos almacenados tanto desde la página web o desde cualquier otro servicio.
- Realizar un control de usuarios tanto en la aplicación web como en los servicios API. Además de llevar a cabo un control de estos según sus roles

e implementar un sistema de sesión de usuario en la parte web.

- Disponer de un sistema de autenticación con aplicaciones externas utilizando la tecnología OAuth2 para facilitar la experiencia de usuario.
- Llevar a cabo un buen tratamiento de la información sensible de los usuarios como pueden ser sus contraseñas, almacenándola de manera segura para evitar filtrados de información.
- Realizar una programación segura teniendo en cuenta algunas de las vulnerabilidades actualmente más comunes. Aquí podemos destacar los controles para evitar SQLInyections o los tokens CSRF para evitar ataques a nuestro servicio.
- Seguir esquemas de desarrollo ágil como en el caso de la arquitectura hexagonal. Esto permite que hayamos conseguido desplegar un proyecto escalable y con gran adaptabilidad frente a los cambios.
- Implementar una automatización en el despliegue de nuestro proyecto a través de Docker, lo que nos permite exportar el proyecto de manera sencilla como por ejemplo a alguna web de hosteo de páginas.

Como vemos, se ha conseguido cumplir con el objetivo establecido para este proyecto, generando una aplicación con gran proyección y que puede servir como punto de partida para proyectos de negocio.

5.2. Trabajos Futuros

Una vez finalizado el proyecto hemos comprobado que se han cumplido todos los objetivos que se definieron en un principio, aunque durante el desarrollo han surgido ideas para futuras implementaciones que pueden ser de gran utilidad. Algunas de estas propuestas podrían ayudar tanto para lo relacionado con los servidores como para la experiencia de usuario.

- **Mayor número de proveedores externos**: introduciendo un mayor número de páginas para la autenticación podríamos mejorar la interacción del usuario dentro de nuestra página. Algunas otras alternativas serian Twitter o Facebook, aunque debemos tener en cuenta que algunos de estos servicios pueden ser de pago por lo que debería ser algo a valorar.
- **Aumentar API**: crear un mayor número de rutas y posibilidades al usuario permite que estos dispongan de una mayor libertad para el desarrollo de aplicaciones externas que dependan de nuestros servicios.

- **Bases de Datos Replicadas**: la creación de bases de datos replicadas consiste en almacenar nuestros datos en diferentes sitios al mismo tiempo para su consulta y edición. Esto permite mayores niveles de seguridad frente a pérdidas de datos, e incluso disminuir el tiempo de respuesta de nuestros servicios en función de la disposición geográfica de los servicios.
- **Incluir Gateway Manager**: un gateway es un intermediario entre el cliente y nuestra API Rest. Una de las ventajas de su implementación sería la facilidad de monetizar nuestros servicios, además de ayudar en el control de usuarios. Un ejemplo sería la utilización de Kong.
- **Monitoreo**: realizar un correcto monitoreo de nuestras APIs permite conocer la cantidad de tráfico que manejamos o los tiempos de respuesta. De esta forma podemos optimizar nuestras peticiones o valorar ciertos cambios en nuestra estructura. Para esto se propone la implementación de Prometheus y Grafana dentro de nuestra arquitectura.

Bibliografía

[1] "Owasp top ten." [Online]. Available: https://owasp.org/www-project-top-ten/

[2] "Why mysql?" [Online]. Available: https://www.mysql.com/why-mysql/

[3] *MongoDB Documentation*. [Online]. Available: https://www.mongodb.com/docs/

[4] *NodeJS Documentation*. [Online]. Available: https://nodejs.org/en/docs

[5] *SpringBoot Documentation*. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[6] *OAuth2 Documentation*. [Online]. Available: https://oauth.net/2/

[7] E. Novoseltseva, "Why mysql?" April 2020. [Online]. Available: https://apiumhub.com/es/tech-blog-barcelona/arquitectura-hexagonal/

[8] "Spring boot oauth2 login with google example." [Online]. Available: https://www.codejava.net/frameworks/spring-boot/oauth2-login-with-google-example

[9] "Api authentication using oauth2 with google," January 2022. [Online]. Available: https://medium.com/geekculture/springboot-api-authentication-using-oauth2-with-google-655b8759f0ac

[10] "Google authorization server." [Online]. Available: https://www.techgeeknext.com/spring-boot-security/google-oauth2

[11] *OAuth apps*. [Online]. Available: https://docs.github.com/en/apps/oauth-apps

[12] "Spring boot oauth2 login with github example." [Online]. Available: https://www.codejava.net/frameworks/spring-boot/oauth2-login-with-github-example

[13] "Basic authentication tutorial with example api," September 2018. [Online]. Available: https://jasonwatmore.com/post/2018/09/24/nodejs-basic-authentication-tutorial-with-example-api

[14] "Using bcrypt-js to hash passwords in javascript," September 2022. [Online]. Available: https://masteringjs.io/tutorials/node/bcrypt

[15] "How to prevent sql injection attacks in node.js." [Online]. Available: https://planetscale.com/blog/how-to-prevent-sql-injection-attacks-in-node-js

[16] "Spring boot upload image," 2023. [Online]. Available: https://www.techgeeknext.com/spring-boot/spring-boot-upload-image

[17] L. Crusoveanu, "Quick guide to spring bean scopes." [Online]. Available: https://www.baeldung.com/spring-bean-scopes

[18] R. Fadatare, "Spring boot user registration and login example tutorial." [Online]. Available: https://www.javaguides.net/2018/10/user-registration-module-using-springboot-springmvc-springsecurity-hibernate5-thymeleaf-mysql.html

[19] "Spring security reference." [Online]. Available: https://docs.spring.io/spring-security/site/docs/5.4.0/reference/html5/

[20] "Spring security architecture." [Online]. Available: https://spring.io/guides/topicals/spring-security-architecture/