
**UNIVERSIDAD POLITÉCNICA SALESIANA**

**SEDE CUENCA**

**CARRERA DE INGENIERÍA DE SISTEMAS**

# IMPLEMENTACIÓN DE UNA PLATAFORMA IOT BASADA EN UNA ARQUITECTURA DE MICROSERVICIOS PARA INVESTIGACIÓN DEL GRUPO GIHP4C Y GITEL EN EL CAMPUS EL VECINO DE LA UNIVERSIDAD POLITÉCNICA SALESIANA

Trabajo de titulación previo a la obtención
del título de Ingeniero de Sistemas

AUTOR: CHRISTIAN ENRIQUE ZHIMINAICELA SEGARRA

TUTOR: ING. GABRIEL ALEJANDRO LEÓN PAREDES, PhD.

Cuenca - Ecuador
2023

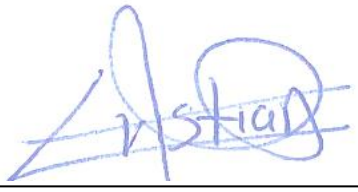
**CERTIFICADO DE RESPONSABILIDAD Y AUTORÍA DEL TRABAJO DE TITULACIÓN**

Yo, Christian Enrique Zhiminaicela Segarra con documento de identificación N° 0106315385, manifiesto que:

Soy el autor y responsable del presente trabajo; y, autorizo a que sin fines de lucro la Universidad Politécnica Salesiana pueda usar, difundir, reproducir o publicar de manera total o parcial el presente trabajo de titulación.

Cuenca, 28 de agosto de 2023

Atentamente,



Christian Enrique Zhiminaicela Segarra
0106315385

**CERTIFICADO DE CESIÓN DE DERECHOS DE AUTOR DEL TRABAJO DE TITULACIÓN A LA UNIVERSIDAD POLITÉCNICA SALESIANA**

Yo, Christian Enrique Zhiminaicela Segarra con documento de identificación N° 0106315385, expreso mi voluntad y por medio del presente documento cedo a la Universidad Politécnica Salesiana la titularidad sobre los derechos patrimoniales en virtud de que soy autor del Proyecto técnico: "Implementación de una plataforma IoT basada en una arquitectura de microservicios para investigación del grupo GIHP4C y GITEL en el Campus El Vecino de la Universidad Politécnica Salesiana", el cual ha sido desarrollado para optar por el título de: Ingeniero de Sistemas, en la Universidad Politécnica Salesiana, quedando la Universidad facultada para ejercer plenamente los derechos cedidos anteriormente.

En concordancia con lo manifestado, suscribo este documento en el momento que hago la entrega del trabajo final en formato digital a la Biblioteca de la Universidad Politécnica Salesiana.

Cuenca, 28 de agosto de 2023

Atentamente,

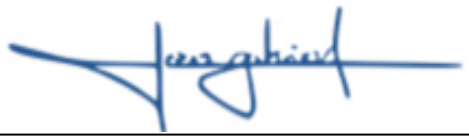
Christian Enrique Zhiminaicela Segarra
0106315385

**CERTIFICADO DE DIRECCIÓN DEL TRABAJO DE TITULACIÓN**

Yo, Gabriel Alejandro León Paredes con documento de identificación N° 0106315385, docente de la Universidad Politécnica Salesiana, declaro que bajo mi tutoría fue desarrollado el trabajo de titulación: IMPLEMENTACIÓN DE UNA PLATAFORMA IOT BASADA EN UNA ARQUITECTURA DE MICROSERVICIOS PARA INVESTIGACIÓN DEL GRUPO GIHP4C Y GITEL EN EL CAMPUS EL VECINO DE LA UNIVERSIDAD POLITÉCNICA SALESIANA, realizado por Christian Enrique Zhiminaicela Segarra con documento de identificación N° 0106315385, obteniendo como resultado final el trabajo de titulación bajo la opción Proyecto técnico que cumple con todos los requisitos determinados por la Universidad Politécnica Salesiana.

Cuenca, 28 de agosto de 2023

Atentamente,



Ing. Gabriel Alejandro León Paredes, PhD.
0103652186

# DEDICATORIA

*Este proyecto se lo quiero dedicar a Dios, por darme salud y fortaleza para cumplir con mis metas, a mis padres Gonzalo Zhiminaicela y María Segarra, por qué día a día lucharon para darme la mejor educación con su gran apoyo durante este proceso. A mis abuelitos Teresa y Alfredo que con su amor incondicional fueron como unos padres para mí, guiándome con buenos valores en el trayecto de la vida. A mi hermana Pao por ser mi compañía en la niñez y por brindarme su aprecio y cariño constante.*

*A mi esposa Mariuchis por ser el complemento fundamental en mi vida, brindándome su compañía en los momentos más difíciles y por apoyarme a terminar mi carrera universitaria. A mis queridos hijos Jesús y Mateo, ellos son y serán mis pilares que me sostienen cada día para poder seguir luchando y cumpliendo las metas que me proponga en mi vida.*

*Que Diosito les proteja toda la vida.*

**Christian Enrique Zhiminaicela Segarra**

## AGRADECIMIENTO

*En primer lugar, quiero agradecer a la vida, a Dios, por darme la sabiduría y el coraje para poder culminar mis estudios universitarios. Agradezco a todos los docentes de la Universidad Politécnica Salesiana, por tan valiosa formación educativa que nos brindaron sin ustedes no sería posible cumplir esta meta. Gracias querida familia por el apoyo, la motivación y el tiempo que me han brindado para educarme y guiarme en el camino de la vida.*

*Un agradecimiento especial a mi director de tesis al PhD. Gabriel León, por brindarme la confianza de creer en mí y darme la oportunidad de realizar mi trabajo de titulación dentro del grupo de investigación GIHP4C, gracias a su paciencia y conocimientos brindados se pudo finalizar este proyecto de la mejor manera posible.*
*Agradezco a todos ustedes por el apoyo brindado, en el transcurso de mi vida universitaria.*

*Que Dios les bendiga.*

**Christian Enrique Zhiminaicela Segarra**

**Resumen**

La tecnología ha evolucionado exponencialmente en los últimos años en todas las áreas, como es la Inteligencia artificial, el blockchain y el paradigma del Internet de las Cosas (IoT), hoy en día IoT ya forma parte de la vida cotidiana, gracias a que la mayoría de las cosas que se encuentran en nuestro entorno se conectan a internet y nos facilita muchas actividades, tal como prender la lavadora, prender la luz con nuestro celular, detectar el nivel del co2, contar con parqueaderos automáticos, y más actividades inteligentes. Todos estos dispositivos pueden ser administrados por plataformas dedicadas a esta tecnología, una de ellas es chirpstack, la cual nos permite administrar las redes de IoT, la misma que se crea con la finalidad de administrar dispositivos inteligentes.

" La arquitectura basada en microservicios según Víctor y Peralta nos dice que ha ganado mucha popularidad por las crecientes demandas de la evolución de los softwares, por su gran facilidad de escalar servicios, tolerante a fallos y permite la incorporación de varias tecnologías para que el software siga evolucionando sin complicaciones." Por lo tanto, los microservicios son una solución estable para incorporar en el desarrollo de una plataforma para dispositivos IoT.

El desarrollo de esta plataforma IoT basada en microservicios, se han centrado en resolver un problema, procesar la data de los nodos de sensores de co2, para poder visualizar mediante gráficas los niveles de co2 que tenemos dentro de la Universidad Politécnica Salesiana. También se dice que esta plataforma nos servirá de guía en diferentes proyectos académicos que involucren el desarrollo de microservicios en la implementación de sistemas para el internet de las cosas.

- Palabras claves: Microservicios, IoT, Cloud, Frontend, Backend, spring boot, Oauth, api gateway, chirpstack, nodo, data.

**Abstract**

Technology has evolved exponentially in recent years in all areas, such as artificial intelligence, blockchain and the Internet of Things (IoT) paradigm. Today, IoT is already part of everyday life, thanks to the fact that Most of the things in our environment are connected to the Internet and it makes many activities easier for us, such as turning on the washing machine, turning on the light with our cell phone, detecting the level of CO2, having automatic parking, and more intelligent activities. All these devices can be managed by platforms dedicated to this technology, one of them is chirpstack, which allows us to manage IoT networks, the same one that is created with the purpose of managing smart devices.

"The architecture based on microservices according to Víctor and Peralta tells us that it has gained a lot of popularity due to the growing demands of the evolution of software, due to its great ease of scaling services, fault tolerant and allows the incorporation of various technologies so that the software continue to evolve without complications." Therefore, microservices are a stable solution to incorporate in developing a platform for IoT devices.

The development of this IoT platform based on microservices has focused on solving a problem, processing the data from the co2 sensor nodes, to be able to visualize the co2 levels that we have within the Salesian Polytechnic University through graphs. We can also say that this platform will serve as a guide for us in different academic projects that involve the development of microservices in the implementation of systems for the Internet of Things.

- Keywords: Microservices, IoT, Cloud, Frontend, Backend, spring boot, Oauth, api gateway, chirpstack, nodo, data.

# ÍNDICE

# Índice de tablas

# Índice de figuras

# Capítulo I

# Introducción

Desde que internet existe, la humanidad ha evolucionado a la par con el crecimiento de las nuevas tecnologías que permiten la comunicación entre las diversas plataformas informáticas, servidores web, así como también, los diversos lenguajes de programación que se han creado han ido incrementando sus funcionalidades de manera que en la actualidad podamos disfrutar de sistemas más robustos, seguros, confiables y escalables que pueden seguir innovando cada vez con una mejor experiencia de uso para el usuario. De la misma manera, se ha ido presenciando muchos avances tecnológicos como el Internet de las Cosas (IoT), que hoy en día han evolucionado tanto que ya forman parte de nuestras vidas. La inclusión de lo expuesto anteriormente forma parte de un concepto conocido como transformación digital, el cual menciona que "es un proceso que se lleva a cabo para actualizar las tecnologías, sus recursos y en definitiva, sus modelos de negocio para no quedarse atrás respecto a la nueva era tecnológica" [1]. La transformación digital ha conllevado que los sectores requieran actualizar sus tecnologías, para tener softwares flexibles y adaptables con las nuevas tecnologías que interconectan dispositivos IoT.

El proyecto tiene como punto clave la implementación de una arquitectura basada en microservicios para la creación de una plataforma IoT, que permite el procesamiento de la información de los nodos de sensores de Co2 por medio de servicios web desarrollados con la tecnología de Spring Boot. Este modelo nos permitir mejorar e incrementar constantemente nuevos servicios dentro de la misma arquitectura para que la plataforma pueda seguir evolucionado y adaptándose a nuevos requerimientos sin importar las tecnologías que se utilicen.

La tesis consta de varios capítulos que describen cada punto de este proyecto, capítulo II que nos describe el problema, donde se presenta los antecedentes, la importancia y el alcance. El tercer capítulo presenta los objetivos del proyecto, tanto el objetivo general y objetivos específicos. El capítulo IV expone los fundamentos teóricos del proyecto con los conceptos investigativos que se realizaron. Seguidamente el capítulo V hace referencia al marco metodológico, donde se explica sobre la metodología del desarrollo de software, así como el levantamiento de los requerimientos, historias de usuario y prototipo. Continuamos con el capítulo VI, donde tenemos el diseño y la implementación de la arquitectura de software para la plataforma IoT, seguido de las herramientas que permiten el despliegue con contenedores docker. El capítulo VII en cambio nos presenta los resultados de la implementación de la arquitectura de software, exponiendo la escalabilidad de los microservicios, la alta disponibilidad, la trazabilidad, el procesamiento de datos de los nodos Co2 y la seguridad del sistema.

# Capítulo II

# Problema

## 2.1 Antecedentes

Actualmente, el grupo de Investigación en Cloud Computing, Smart Cities & High Performance Computing (GIHP4C) disponen con una plataforma web que gestiona los datos recogidos por la red de sensores de IoT que se encuentran distribuidos por el campus universitario, esta plataforma usa tecnologías antiguas con una arquitectura monolítica, que no permite hacer actualizaciones constantes sin tener que parar el servicio y la escalabilidad es más compleja aún, ya que no nos permite el desarrollo continuo sobre dicha plataforma web para la gestión de una red de sensores IoT.

## 2.2 Importancia

Con base a lo antes expuesto, la propuesta que realizamos es reemplazar dicha arquitectura preexistente con nuevas tecnologías, integrando una nueva arquitectura basada en microservicios, así como también la implementación de contenedores para el despliegue de estos, de esta manera queremos lograr que nuestro sistema nos permita la escalabilidad y posteriormente pueda ser tomado como base para futuros proyectos relacionados con la misma rama.

## 2.3 Alcance

La plataforma IoT que se desarrollará en este proyecto será utilizada por los estudiantes de la Universidad Politécnica Salesianas, en concreto por el grupo de Investigación en Cloud Computing, Smart Cities & High Performance Computing (GIHP4C) y por el grupo de Investigación en Telecomunicaciones y Telemática (GITEL) de la Universidad Politécnica Salesiana, para realizar análisis de los nodos de sensores que se encuentran dentro del campus universitario.

# Capítulo III

# Objetivos Generales Específicos

## 3.1 General

Implementación de una plataforma IoT basada en una arquitectura de microservicios para investigación del grupo GIHP4C y GITEL en el Campus el vecino de la Universidad Politécnica Salesiana.

## 3.2 Específicos

- Estudiar sobre arquitecturas de microservicios y contenedores orientados a la gestión de plataformas IoT.
- Establecer los requisitos funcionales y no funcionales para la plataforma a desarrollar.
- Desarrollar una aplicación web para la visualización de datos de sensores IoT.
- Desplegar la plataforma web en los servidores del grupo de Investigación GIHP4C utilizando entornos de contenedores.
- Redactar el documento que respalde las memorias del proyecto.

# Capítulo IV

# Revisión de la literatura o fundamentos teóricos

## 4.1 Cloud Computing

La computación en la nube representa un modelo que permite el acceso a la red Global, que contiene recursos informáticos configurables y compartidos como; redes, servidores, base de datos, aplicaciones y servicios que se pueden aprovisionar y liberar rápidamente con un mínimo esfuerzo de gestión o interacción con el proveedor de servicios [2].

Según [3] "La computación en la nube, nos brinda una plataforma para el acceso bajo demanda a varios recursos informáticos con mayores capacidades en términos de almacenamiento y potencia de procesamiento, lo que permite el ingreso a los servicios en la nube desde cualquier lugar y en cualquier momento"

### 4.1.1 Servicios de computación en la nube

En base al autor [2] una arquitectura de nube se puede dividir en el backend y el frontend. La interfaz se hace visible para el usuario a través de conexiones a Internet, y el backend comprende los diversos modelos que son:

- **Software como servicio (SaaS):** En [4] nos dice que SaaS ofrece software de aplicación a través de Internet, para que varios consumidores de la nube accedan

simultáneamente a través de una conexión a Internet. Se ofrece al usuario un conjunto alojado de software que se ejecuta en una plataforma e infraestructura propiedad del proveedor de servicios en la nube, sin preocuparse por la compleja gestión de software y hardware [2]. En [4] nos declaran los siguientes ejemplos "OneDriver, Google Drive, Dropbox, Software CRM y Cisco WebEx."

- **Plataforma como servicio (PaaS):** PaaS "es un servicio de desarrollo a través de internet, el usuario no requiere ningún requisito de instalación de software o hardware, también tiene un software que incluye una base de datos, middleware y herramientas de desarrollo" [2]. El cliente en la nube gestiona el software y los servicios de aplicaciones, algunos ejemplos de PaaS son: Azure App Service, AWS Elastic Beanstalk y Google App Engine [4].

- **Infraestructura como servicio (IaaS):** Los autores en [2], Afirman que IaaS permite almacenar datos en diferentes ubicaciones geográficas, también controlan las actividades en los centros de datos en la nube y permiten a los usuarios la flexibilidad de implementar y gestionar los mismos servicios de software para sus necesidades empresariales, como middleware, entornos en tiempo de ejecución, bases de datos y software de aplicaciones. Algunos ejemplos de IaaS son Microsoft Azure, Amazon Web Services (AWS) y Google Compute Engine (GCE) [4].

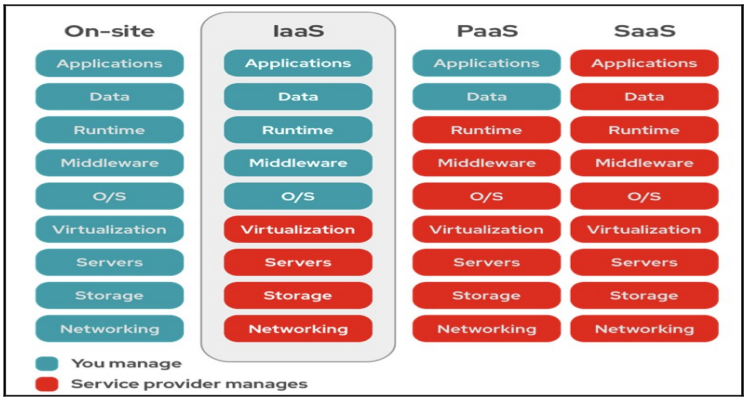



Figura 4.1: Diferencias entre Servicio IaaS, PaaS y SaaS.
[5]

### 4.1.2 Tipos de computación en la nube

El autor en [2], declara las siguientes nubes:

- **Nube pública:** Este tipo de nube son implementadas por organizaciones de pago por uso, ya sea por suscripciones mensuales o anuales, está basado en la oferta de servicios de computación virtualizados por parte de los proveedores para múltiples usuarios [5]. Las nubes públicas ofrecen recursos compartidos y son altamente escalables. Algunos ejemplos de proveedores de nube pública conocidos son Microsoft Azure, Amazon Web Services (AWS) y Google Cloud [4].

- **Nube Privada:** En [4], los autores indican que la nube privada permite establecer centros de datos internos de una organización. Según [5], los recursos ofrecidos pasan a ser propiedad de una sola institución, manteniendo la infraestructura en una red privada de uso, y llegando incluso a ofrecer la posibilidad de alojar los servicios en las propias instalaciones del cliente. El autor en [2], indica que las nubes privadas tienen una escalabilidad limitada y están restringidas a un área.

- **Nube Híbrida:** Se refiere a la combinación de recursos de la nube privada y la nube pública, "son utilizadas cuando la corporación necesita aumentar los recursos privados de dichos servicios, por lo cual la corporación necesita contratar servicios de una nube pública para la expendición de los recursos de la nube privada". El autor en [2], declara que las actividades principales se alojan en una nube privada, mientras que los servicios menos esenciales se subcontratan a una nube pública. Cada una de las nubes sigue siendo una entidad única, pero vinculada entre sí y están sujetas a problemas de red y seguridad.

### 4.1.3 Plataformas de Cloud Open Source

Como hemos mencionado en los conceptos anteriores, tenemos varias formas de almacenar la información mediante el cloud computing, nosotros nos enfocamos en analizar las nubes de open Source:

- **OPENSHIFT:** Este tipo de nube son implementadas por organizaciones de pago por uso, ya sea por suscripciones mensuales o anuales, está basado en la oferta de servicios de computación virtualizados por parte de los proveedores para múltiples usuarios [5].

- **OPENNEBULA:** Es un servicio de código abierto, que nos brinda una gestión más sencilla, nos permite el compartimiento de recursos. Dispone de las últimas actualizaciones en el campo de la virtualización de data centers para un desarrollo básico, que permite desplegarlo en una infraestructura, volúmenes, capacidades preexistentes y además no depende de hypervisores [6]. Además, facilita la creación de cualquier tipo de nubes: públicos, privados e híbridos, que como componente adicional en una nube híbrida realiza una estructura propia de infraestructura pública y privada que este obtiene a partir de proveedores públicos como es AWS, google y ElasticHost, es decir que gracias a estas plataformas se puede tener escalabilidad y alta disponibilidad [6].

- **CLOUDSTACK:** Como una plataforma open source con características esenciales que la mayoría de las empresas demandan al desplegar un Cloud como servicio donde su uso general es para desplegar infraestructuras (IaaS) como las más conocidas tenemos las Cloud privadas, públicas e híbridas basándose en conjuntos de recursos informáticos [7]. Es decir, esta plataforma permite gestionar su cómputo, red (NaaS), almacenamiento y gestión de cuentas aprovechando tanto el hardware como el software del equipo. Una característica común con plataformas similares es que utiliza hipervisores conocidos como KVM, vSphere y XenServer / XCP para desplegar su virtualización teniendo también una compatibilidad con las APIS de AWS EC2 y S3 [7].

- **OPENSTACK:** Es un servicio de código abierto “que hace uso de varios recursos virtuales para gestionar y diseñar los tres tipos de nubes públicas, privadas e híbridas”. La plataforma OpenStack “en la virtualización, los recursos, como el almacenamiento, la CPU y la RAM, se extraen de distintos programas específicos de los proveedores y se dividen con un hipervisor antes de distribuirlos según sea necesario. OpenStack utiliza un conjunto uniforme de interfaces de programación de aplicaciones (API) para extraer todavía más recursos virtuales, los cuales distribuye en conjuntos distintos que se utilizan para potenciar las herramientas del Cloud Computing estándares que utilizan los administradores y los usuarios” [8].

## 4.2 Software Web

Según [9], las aplicaciones web se separan en dos partes: backend que hace referencia al servidor, que es responsable del procesamiento de datos, y frontend "cliente", el que consume los servicios del backend y proporciona una interfaz conveniente para la interacción entre usuarios y la aplicación web.

### 4.2.1 BackEnd

Se considera como una capa de acceso a datos que se conecta con bases de datos y manipula la información para que sea accesible mediante una API generada. Según [9] los componentes de backend podrían desarrollarse utilizando diferentes enfoques arquitectónicos, uno de ellos Microservicios, que es el más adecuado para sistemas escalables. La idea del enfoque se basa en el concepto de separación de partes lógicamente independientes de un sistema, remplazando a los estilos arquitectónicos monolíticos porque son difíciles de escalar y de desarrollar diferentes partes simultáneamente.

Para este desarrollo existen varias Frameworks como “PHP (Laravel), Java (Spring), Python (Django), Ruby (Ruby on Rails), Express JS”, entre otras, nosotros nos enfocáremos en la herramienta JAVA (Spring Boot) para desarrollar nuestro Backend.

**Spring Boot**

El artículo [10], nos dice que Spring Boot es una extensión de Spring Framework que sigue el enfoque de "Convención sobre configuración", que no permite construir aplicaciones basadas en Spring de manera rápida y fácil. En [10], explica que “el objetivo principal de Spring Boot es simplificar el desarrollo de aplicaciones por medio de la autogestión de un gran número de configuraciones, tareas y componentes que son necesarios para la ejecución de un sistema”. El autor [10], declara las siguientes características de Spring Boot:

- Posee servidores de aplicaciones y contenedores de Servlet embebido.
- Nos permite configurar al sistema con archivos .properties o .yml. Y no por código o XML.
- Gracias a las dependencias tenemos “configuraciones automáticas para diferentes Frameworks como es Spring Security, Spring Cloud Gateway, Spring Batch y otros”.
- Nos permite crear “aplicaciones basadas en arquitecturas de microservicios”.

### 4.2.2 FrontEnd

EL autor en [11], declara que el frontend es la parte con la que el usuario interactúa con las páginas web mediante menús, formularios de contacto, multimedia y más, para tener acceso a esas páginas necesitamos usar un navegar web ya sea firefox, chrome, safari y otros. Para diseñar y desarrollar la interfaz web "frontend", se utilizarán “tecnologías, que son una

variación de HTML, CSS y JavaScript, todos controlados por el navegador". Para este desarrollo existen varios frameworks como React, Angular, Vuejs, Emberjs, entre otros, nosotros nos enfocáremos en el framework Angular para desarrollar nuestro Frontend.

**Angular**

Es un framework de desarrollo web frontend que funciona como un marco de JavaScript que ofrece una sólida colección de componentes que simplifican el arte de escribir, alterar y usar código. Este marco de código abierto fue desarrollado por Google bajo la licencia MIT en 2010. Angular nos ayuda a desarrollar aplicaciones escalables de sitios web que se basan en contenido dinámico y que quieran ofrecer un tiempo de carga más rápido y una mayor seguridad. Nos permite reutilizar su código y habilidades para crear aplicaciones para cualquier objetivo de implementación. Para web, web móvil, móvil y escritorio nativo [12].

## 4.3 Contenedores

El autor en [30], indica que el contenedor "es un término genérico utilizado para designar una caja que transporta mercancías, y así poder aislar las diferentes cargas suficientemente resistentes para su reutilización, habitualmente apilable y dotado de elementos para permitir la transferencia entre modos de transporte, ya sea de forma aérea, terrestre o marítima".

### 4.3.1 Virtualización

La virtualización hace referencia a la creación de un nuevo entorno no físico "virtual" mediante un programa que se instala en una máquina física, esto simula los componentes físicos de un computador para poder ejecutar varias instancias de sistemas operativos dentro de uno solo. Este proceso de virtualización se realiza gracias a un software llamado hipervisor, este software nos ayuda a inicializar diversos sistemas operativos al mismo tiempo.

### 4.3.2 Contenedor de Sotfware

Las tecnologías de desarrollo de software hoy en día usan la misma logística del transporte internacional, un contenedor dentro de software hace referencia a una agrupación de paquetes que engloba código de una aplicación con sus respectivas bibliotecas, archivos y dependencias necesarias para poder ejecutarse en cualquier entorno. Esto nos permite acelerar los procesos de desarrollo de software y su despliegue en ambientes productivos.

### 4.3.3 Dockers

Se trata de un proyecto de código abierto "open source" el cual tiene por objetivo compilar y ejecutar aplicaciones sobre contenedores de software, separa las aplicaciones de la infraestructura y esto nos permite agilizar el proceso de despliegue de software y actualizaciones constantes sin tener que bajar el servicio, esto reduce de forma significativa el tiempo desde que se escribe el código de la aplicación hasta que esta se pone en producción [13].

Docker cuenta con un ciclo de vida fácil para poder desarrollar de manera más rápida las aplicaciones:

- **Construir:** Permite desarrollar de manera eficiente sus propias aplicaciones empaquetando las aplicaciones como imágenes de contenedores portátiles para ejecutarse en cualquier entorno y funciona con todas las herramientas de desarrollo.
- **Compartir:** Nos permite compartir y utilizar imágenes propias del repositorio de Docker Hub.
- **Ejecutar:** Nos permite el despliegue de las aplicaciones en contenedores separados de forma independiente y en diferentes idiomas.

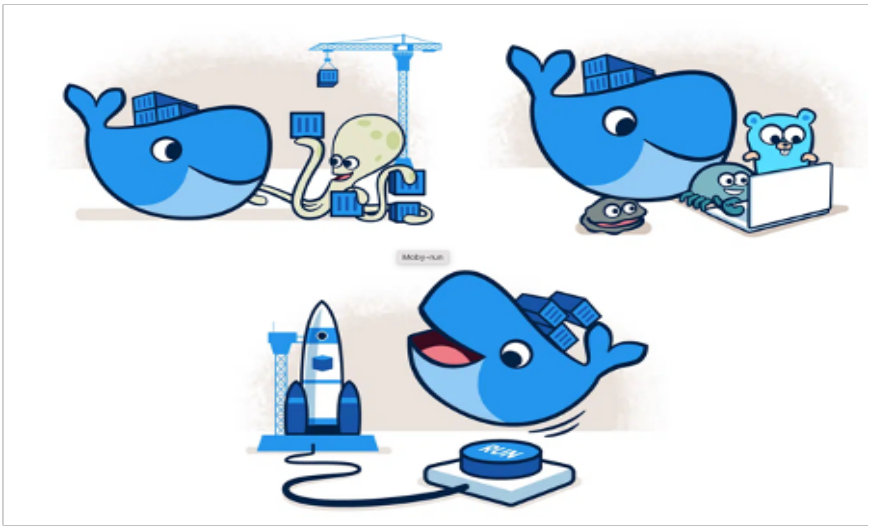


Figura 4.2: Ciclo de vida Docker. [13]

### 4.3.4 Arquitectura Docker

Según los autores, en [13] docker "utiliza una arquitectura cliente servidor, en la que tanto el cliente como el servidor pueden funcionar en el mismo sistema", o en máquinas diferentes, ya que estos usan una API REST para comunicarse entre sí, mediante un cliente docker o demonio docker remoto.

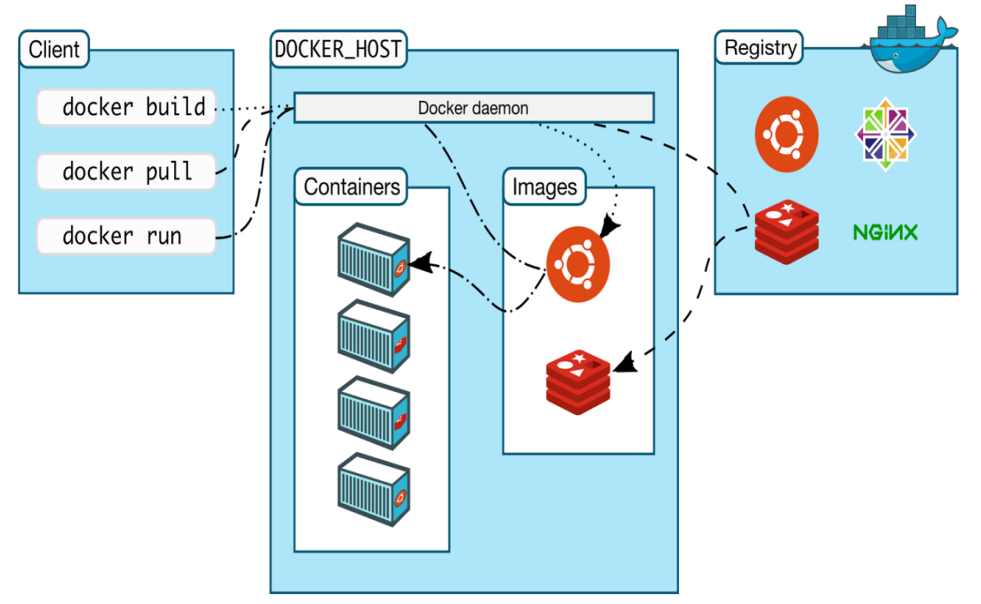



Figura 4.3: Arquitectura Docker.
[13]

- **El Demonio Docker:** Este demonio se comunica con otros demonios para gestionar objetos docker como imágenes, contenedores, redes y volúmenes, este demonio escucha las peticiones mediante la API Docker [13].

- **El cliente Docker:** Este cliente se comunica con uno o más demonio docker utilizando la API Docker, mediante los comandos docker run [13].

- **Escritorio Docker:** El autor en [13], indica que la aplicación se descarga y se instala en cualquier sistema operativo, permitiéndonos compartir, crear aplicaciones y microservicios en contenedores y ya tiene incluido el demonio.

- **Registros de Docker:** El registro docker se utiliza para almacenar y buscar las imágenes en docker hud y poder descargarlas desde cualquier entorno, aquí se tiene registros públicos o privados [13].

- **Imágenes** Hace referencia a un archivo de lectura que contiene las indicaciones para crear un nuevo contenedor, ya sea basado en linux u otro entorno, pero que contiene todas las dependencias necesarias para poder ejecutar un aplicativo. "También se puede crear imágenes propias mediante un DockerFile con las configuraciones necesarias para que se pueda ejecutar, si tenemos algún cambio en el dockerFile solo se reconstruye las capas que fueron cambiadas, de esta manera las imágenes que se utilizan o se crean son más ligeras y rápidas en comparación con la virtualización" [13].

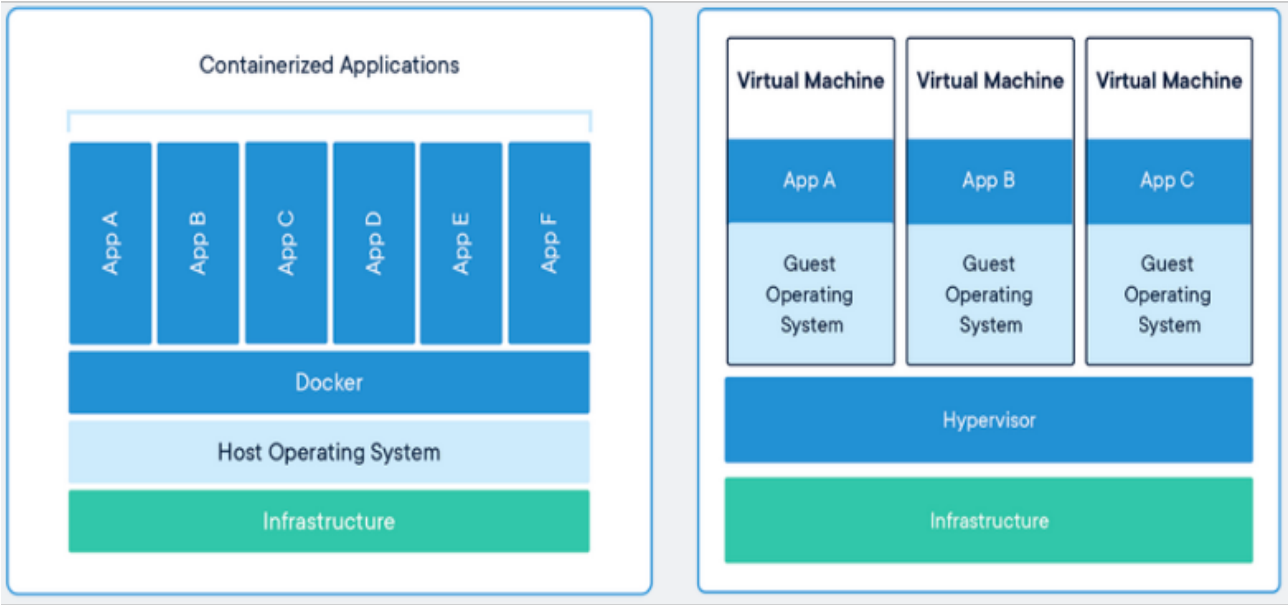



Figura 4.4: Contenerización vs Virtualización.
[13]

## 4.4 Microservicios

Es un conjunto de pequeños servicios independientes que cooperan entre sí mediante protocolos livianos de comunicación, estos servicios se construyen y despliegan de forma individual, y usualmente se apoyan en procesos de integración y despliegue continuo de forma autónoma y automatizada [14].

Los microservicios nos permiten un acceso único mediante la API gateway, para poder acceder a los recursos de cada microservicio dependiendo de la petición realizada. El autor en [15] nos redacta que la mayoría de los microservicios implementan el patrón Circuit Break

que provee de una eficiente tolerancia a fallos, y nos ofrecen solución a problemas comunes en distintas aplicaciones facilitando su reutilización.

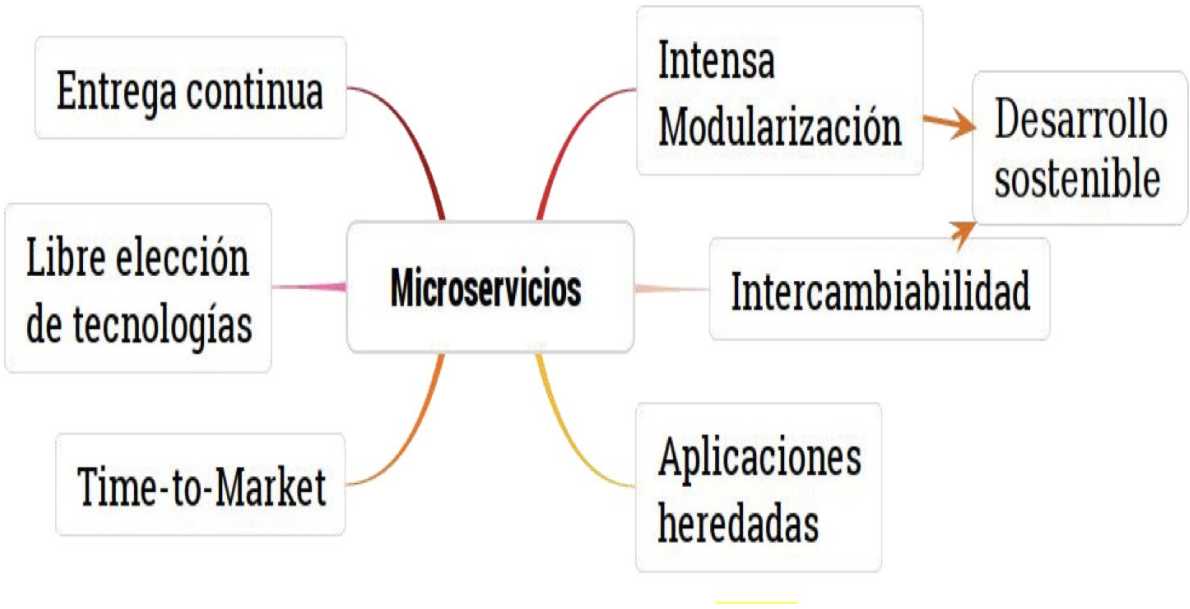



Figura 4.5: Beneficios de los microservicios
[15]

Los microservicios cambian el enfoque tradicional de una arquitectura monolítica, porque nos permite escalar las aplicaciones distribuyendo los servicios o componentes en los servidores y así pueden escalar solo los que sean necesarios, sin tener que escalar el sistema entero, cada nuevo servicio se puede desarrollar usando diferentes tecnologías y persistencia de datos. En la figura 4.6 se observa los cambios que obtiene una arquitectura de microservicios.

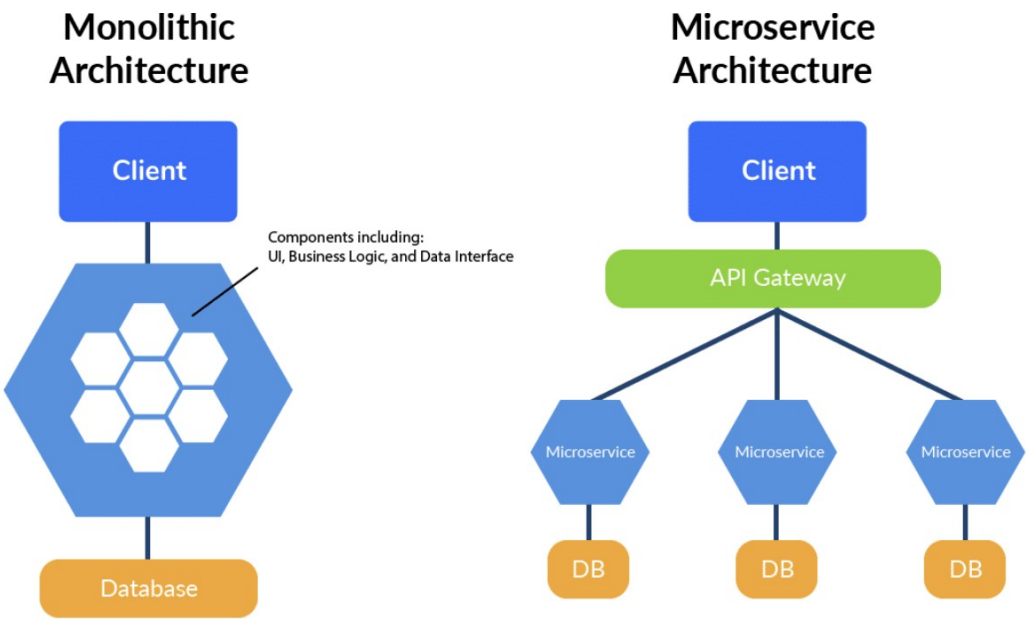



Figura 4.6: Monolítica vs Microservicios
[16]

## 4.5 Internet de las Cosas (IoT)

El IoT no es más que la interconexión de objetos, cosas, elementos cotidianos a la Internet, a través de diferentes tecnologías como son los dispositivos habilitados con la tecnología inalámbrica abierta como bluetooth, zigbee, radiofrecuencia, Wi-Fi y los servicios que brindan teléfonos inteligentes, así como sensores, ya sea de temperatura, ruido y actuadores que están integrados a los objetos. La ventaja del IoT es que se desarrolla utilizando muchos de los estándares de Internet existentes, para ofrecer servicios de transferencia de información y análisis de datos, así como el desarrollo de aplicaciones que interactúen y se comuniquen con los usuarios mostrando la información sobre el entorno que nos rodea, mediante una aplicación [17].

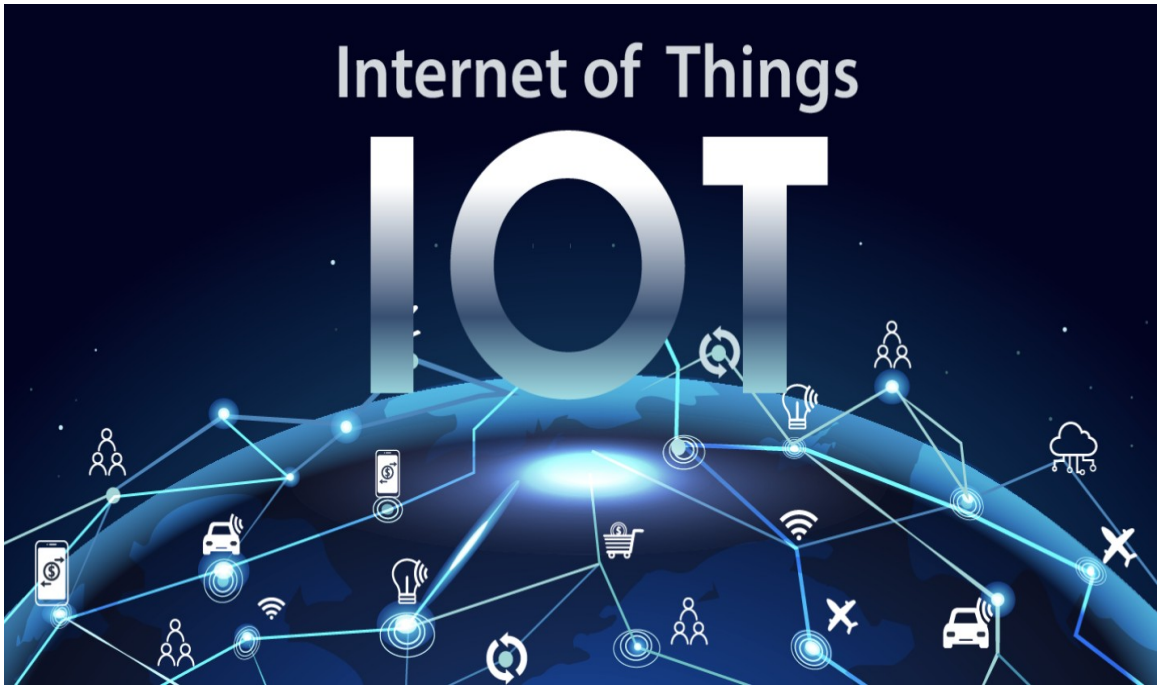



Figura 4.7: Aplicaciones IoT
[18]

### 4.5.1 Arquitectura del Internet de las Cosas (IoT)

Los autores en [19] y [20] definen a una arquitectura IoT como una colección de objetos físicos, sensores y actuadores con capacidad de comunicación bajo estándares y protocolos IoT que funcionan en base a determinadas capas. "Las capas de IoT son fragmentos funcionales del sistema IoT, cumpliendo funciones capaces de identificar y localizar objetos de manera inteligente en tiempo real, mediante el uso de componentes que permiten la recolección de datos y envío a través de la nube o red para su procesamiento y almacenamiento".

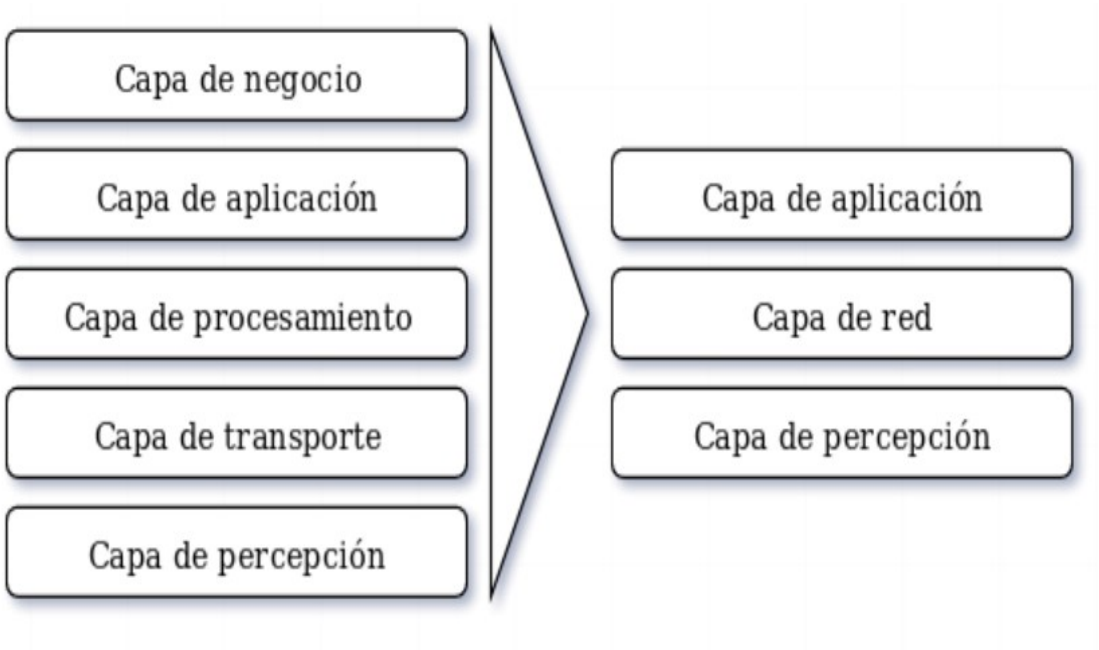



Figura 4.8: Arquitectura IoT
[19]

La figura 4.8 muestra la arquitectura IoT convencional basada en cinco capas. Esta arquitectura permite ser escalable y flexible para adaptarse a diversos componentes y tecnologías, estas capas son descritas seguidamente como afirman en [19],[20] y [21]

- **Capa de negocio:** Capa que realiza la administración general de todas las actividades y servicios de IoT.
- **Capa de aplicación:** La capa "proporciona servicios de aplicación específicos a usuarios".
- **Capa de procesamiento:** Hace referencia a un "middleware que nos ayuda a almacenar, enviar y procesar los datos, puede utilizar varias tecnologías como bases de datos, procesamiento de Big Data y otros módulos".
- **Capa de transporte:** El objetivo de esta capa es "transportar los datos de la capa de percepción hacía la capa de procesamiento y viceversa a través de tecnologías como:

bluetooth, 3G, 4G, Local Area Network (LAN), Near Field Communication (NFC) y Radio Frequency Identification (RFID)" [19].

- **Capa de percepción:** Dentro de esta capa física se "procede a la detección de información, ambiente, parámetros físicos y otros objetos inteligentes".

Los autores en [19],[20] y [21] nos redactan que la "arquitectura puede definirse en tres capas, abstrayendo la capa de negocio dentro de la capa de aplicación, así como la capa de trasporte y procesamiento dentro de la capa de red, el cual define la idea general de todo IoT".

### 4.5.2 Protocolos de Comunicación

Cada dispositivo IoT interactuar entre sí para la cual esta intercomunicación se realiza mediante el uso de protocolos estandarizados en la industria, a continuación, realizaremos un breve análisis de algunos de ellos para conocer más sobre estas tecnologías. "Los protocolos de comunicación que más se han usado son HTTP, MQTT y CoAP, para la parte de la comunicación existen varias redes como LoRa o SigFox. Son redes WAN para el IoT y una alternativa a los sistemas tradicionales de comunicación" [22].

**CoAP**

CoAP (Protocolo de aplicación restringida) es un protocolo de transferencia de mensajes estandarizados. "Está destinado a dispositivos con recursos limitados, fue diseñado bajo el paradigma petición/respuesta y proporciona un modelo de intercambio de mensajes para transferir datos de sensores como temperatura, humedad y ubicación, en arquitecturas de red tipo REST, permitiendo fácilmente su traducción a HTTP. Esto permite la integración de datos de sensores en servicios basados en web aportando valor al ecosistema IoT" [23].

El autor en [24], declara las siguientes características más relevantes del protocolo de comunicación para IoT según el Estudio de Mapeo Sistemático.

**Características**

- El protocolo CoAP permite “la transmisión de datos manteniendo el tamaño del mensaje lo más pequeño posible y admite el mecanismo de retorno de la espera” [24].
- Hace uso de “(URI) en lugar de encabezados, sin embargo, los usa según el servidor de la aplicación” [24].
- Permite utilizar mensajes “válidos o inciertos para proveer dos niveles diferentes de calidad de servicio. Allí, los mensajes verificados son recibidos por el destinatario y los mensajes no verificados” [24].
- Es considerado como “un reemplazo de HTTP para las redes de IoT” [24].

**MQTT**

Las siglas significan “(Transporte de telemetría de cola de mensajes) se trata de un protocolo de mensajes originalmente desarrollado en 1999”, fue diseñado como un transporte de mensajería de publicación/suscripción “con el objetivo de proponer un protocolo de mensajes ligero, de bajo consumo energético y empleando el mínimo ancho de banda” [23]. MQTT está basado en sesiones, esto quiere decir que, tras establecer la conexión TCP, el proceso completo de comunicación se divide en cuatro etapas, creación de la conexión MQTT, autenticación, comunicación y terminación de la sesión [23].

El autor en [24], declara las siguientes características más relevantes del protocolo de comunicación para IoT según el Estudio de Mapeo Sistemático.

**Características**

- Es considerado “ideal para dispositivos de recursos comprimidos los cuales ocupan enlaces con menor confiabilidad y de bajo ancho de banda” [24].
- Es utilizado “como un protocolo de comunicación apropiado para IoT y M2M” [24].
- MQTT tiene “como requisito clave utilizar un ancho de banda bajo para enviar datos y requisitos para un recurso de dispositivo pequeño” [24].

**HTTP**

HTTP (Protocolo de Transferencia de Hipertexto) “es un protocolo de nivel de aplicación basado en una arquitectura cliente-servidor frecuentemente utilizado en servicios web, fue diseñado bajo el paradigma petición/respuesta, proporciona un modelo de intercambio de datos entre cliente y servidor basado en peticiones” [23].
HTTP define “los métodos GET, PUT, POST o DELETE mediante los cuales el cliente puede interactuar solicitando datos, actualizarlos o borrándolos respectivamente en un servidor”[23].

El autor en [24] declara las siguientes características más relevantes del protocolo de comunicación para IoT según el Estudio de Mapeo Sistemático.

**Características**

- HTTP “usa el identificador de recursos universal (URI)”.
- HTTP “se integra con REST se ha utilizado en la arquitectura IoT”.
- HTTP “utiliza cuatro modos, como POST, GET, PUT y DELETE eficientes para IoT”.
- HTTP “utiliza muchos anchos de banda del modelo de solicitud y respuesta”.
- HTTP permite “enviar archivos de gran tamaño para IoT”.

**REST**

REST “es un estilo arquitectónico ligero para el uso de servicios web, es accedido mediante una URI, pero con una notación y formato JSON bajo el funcionamiento de la capa de aplicación, que frente a XML lo hace más liviano, REST tiene una adecuación liviana para el intercambio de información con clientes IoT como se muestra en la figura 4.9”. En [25] la figura muestra las peticiones que hace cada cliente hacía los servidores, “el cual intercambian información controlada por códigos de estado HTTP” [25].

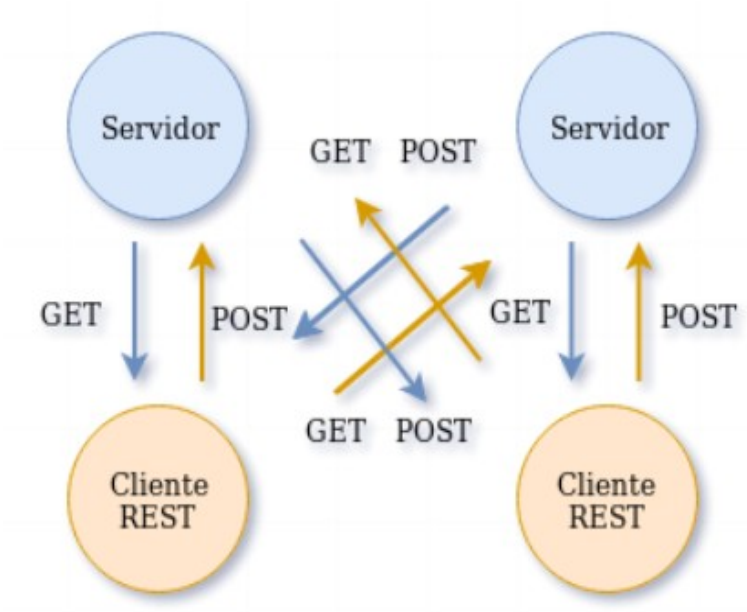



Figura 4.9: Modelo de operación REST.
[25]

**LORAWAN**

El autor en [29] declara que LoRaWAN “es una red de área ancha inalámbrica de baja potencia (LPWAN) que apunta a los requisitos clave del Internet de las cosas tales como los servicios de comunicación bidireccional, de movilidad y de localización, todos ellos de manera segura. Esta norma facilita la interoperabilidad sin fisuras entre los dispositivos inteligentes”, también nos ofrece comunicación inalámbrica con un rango de transmisión bastante amplio y su consumo eléctrico es bastante bajo, cuenta con una interfaz de comandos ASCII sobre UART.

La arquitectura de red presenta “una topología en estrella en la que las puertas de enlace son un puente transparente de transmisión de mensajes entre dispositivos y un servidor de red central. Las puertas de enlace están conectadas al servidor de red a través de conexiones IP estándar, mientras que los dispositivos utilizan la comunicación inalámbrica de un solo salto a una o muchas puertas de enlace, también se basa en el uso de nodos, puertas de enlace que de manera similar a los puntos de acceso Wi-Fi recogen las señales del aire que sirve eficazmente como puente de datos a la aplicación”, permite que los datos se transmitan a los sensores [29].

# Capítulo V

# Marco metodológico

En el siguiente capítulo presentaremos la metodología de desarrollo que hemos utilizado para el desarrollo del proyecto.

## 5.1 Metodología de Desarrollo de Software

### 5.1.1 Introducción

Con la “metodología Scrum lo que se pretende es alcanzar el mejor resultado de un proyecto determinado, nos explica que las prácticas que se aplican con la metodología Scrum sé retro alimentan unas con otras y la integración de estas tiene su origen en un estudio de cómo hay que coordinar a los equipos para ser potencialmente competitivos” [26].

### 5.1.2 Marco de trabajo de Scrum

Marco de trabajo de Scrum se dividen en las siguientes fases:

- **El qué y el quién:** Dentro de los miembros se identificarán los roles y la responsabilidad de estos.
- **El cuándo y dónde:** Es un “sprint como tal”.
- **El cómo y el porqué:** Hace “referencia al equipo de Scrum y así mismo a la herramienta que se van a usar”.

### 5.1.3 Roles de Scrum

**Product Owner**

Es la "persona encargada de financiar el proyecto, y también de realizar la publicación necesaria para que el proyecto sea eficiente".

**Scrum Master**

Es la persona "encargada de asegurar que las bases y técnicas de la metodología están siendo aplicadas para el correcto control del desarrollo del proyecto".

**Stakeholders**

Son los actores que se interesan en el desarrollo del sistema y que este funcione de manera correcta.

**Equipo de desarrollo**

Son "los encargados de realizar las pruebas y el desarrollo del producto".

### 5.1.4 Descripción del Scrum

El autor en [31], indica que para "el proyecto planteado seguirá la metodología de desarrollo SCRUM de la familia de procesos ágiles para la gestión y control de proyectos, que no es más que un marco de trabajo diseñado para lograr la colaboración eficiente de equipos en proyectos empleando reglas, artefactos". Además, "define roles que generan la estructura necesaria para su correcto funcionamiento. Scrum define un Sprint que hace referencia a ventana de tiempo donde se crea una versión utilizable del producto. Cada sprint es considerado como un proyecto independiente y su duración máxima es de un mes. Además, el sprint definido se compone de una reunión de planeación del Sprint, Daily Scrum, trabajo de desarrollo, revisión del sprint y retrospectiva del Sprint" [31].

A continuación, se enumeran los principales aspectos como metodología ágil.

- Se considera una forma de desarrollo adaptable.
- Está básicamente orientado a personas principalmente.
- Emplea "el modelo de construcción incremental basado en iteraciones (sprint) y revisiones".
- Es indicada para proyectos con requisitos de cambio rápido.

En [32] afirma que existen etapas comunes de cada Sprint mencionadas a continuación:

- **Reunión de planificación del Sprint: "**Aquí se divide el tiempo de duración del Sprint, así como el objetivo y entregable del mismo, teniendo en cuenta que el equipo de desarrollo deberá saber cómo realizarlo". Cada uno de los involucrados nos fijaremos un objetivo en concreto que tendremos que llegarlo a cumplir durante el tiempo que definiremos para el sprint, por lo general cada sprint tendrá una duración de 15 días, en casos de que las actividades involucren alguna complejidad podríamos planificarlo para más tiempo, siempre y cuando poniéndonos de acuerdo con todo el equipo. Además, en esta fase cada persona que se encuentra implicado dentro del proyecto se autoasignará una o varias actividades que cree poder desarrollar y colocarse un nivel de complejidad según la tarea que escoja, dicho nivel de complejidad se basará en una regla de Fibonacci, siendo el número 1 el menos complejo y así sucesivamente.

- **Scrum diario:** Hace referencia a la sincronización de las actividades para la elaboración del plan del día. Esta actividad comprenderá una reunión diaria con una duración de 20 minutos en la que tendrá participación todos los involucrados en el proyecto, en dicha reunión expondremos las actividades que estamos llevando a cabo para alcanzar los objetivos del sprint en progreso, así como también las actividades que desempeñaremos

en el trayecto del día, todo enfocado en cumplir los objetivos propuestos anteriormente durante la planificación del Sprint.

- **Trabajo de desarrollo durante el sprint:** Se asegura que "el objetivo se esté cumpliendo, sin cambios que puedan tomar otro rumbo dicho objetivo del Sprint y se mantiene un feedback (retroalimentación) constante con el cliente o dueño del proyecto. Cada involucrado dentro del proyecto llevara a cabo sus actividades para conseguir los objetivos planteados, cabe destacar que también en esta fase no se efectuaran ningún cambio de último momento dentro del proyecto, ya que podríamos afectar a los objetivos de este" [33].

- **Revisión del Sprint:** Es la reunión que se realiza con el cliente, en donde "se estudia y revisa el Product Backlog (listado de requerimientos) del sprint. En esta reunión evalúan cual es la situación actual del producto en desarrollo y también se llegan a definir los cambios a realizar, para un posterior sprint. Esta reunión se desarrollará siempre al final de cada sprint de manera en la cual el equipo de desarrollo exponga sus incrementos de creación del producto final" [34].

- **Retrospectiva del proyecto:** Es en donde "el equipo de desarrollo tiene la oportunidad de mejorar su proceso de trabajo y aplicar los cambios en los siguientes sprints". Por lo general esta reunión se realiza junto con la planificación del sprint posterior, en este punto se estima lo que hemos llevado a cabo y lo que podríamos mejorar posteriormente en futuros sprints, se lleva una retroalimentación para no cometer errores en los cuales hemos fallado anteriormente.

En la Tabla 1. tabla definiremos los involucrados en este proyecto, con sus respectivos roles.

| **Roles** | **Actor/Actores** |
|---|---|
| **Scrum Master** | Dr. Gabriel Alejandro León Paredes |
| **Product Owner** | Ing. Juan Inga Ortega |
| **Equipo de Desarrollo** | Est. Christian Enrique Zhiminaicela Segarra. |

Tabla 5.1: Roles de Implementación en la metodología

## 5.2 Levantamiento de Requisitos

En [36] indica que los requisitos son “una característica que se debe exhibir por el software desarrollado o adoptado para solucionar un problema particular”. Los requisitos se enfocan en dar solución a problemas de la cotidianidad o en otro aspecto para apoyar en procesos de automatización de tareas que un usuario requiere. Otro aspecto para tomar en cuenta dentro del área de requisitos, estos sirven para brindar soporte dentro de actividades y procesos de negocios de usuario incluso para hacer la corrección de algún producto previamente construido.

Tomando en cuenta que la aplicación a construir debe tener como referencia un punto de partida, con la licitación de requerimientos se busca llegar a obtener una definición en breve de las partes interesadas de que es lo que se tiene que llegar a construir como tal, con definiciones o "una descripción de una condición o capacidad que debe cumplir un sistema, arquitectura o implementación, ya sea derivada de una necesidad de usuario identificada, o bien estipulada en un contrato, estándar o especificación u otro documento formalmente impuesto al inicio del proceso" [35].

### 5.2.1 Alcance

Esta herramienta se utilizará por los diferentes estudiantes y docentes involucrados en procesos de investigaciones referentes al IoT y automatización de procesos dentro de los grupos de investigación en GITEL y GIHP4C de la Universidad Politécnica Salesiana.

### 5.2.2 Personal Involucrado

Dentro de la planificación del proyecto se encuentran involucradas varias personas que tienen diferentes roles o finalidades en este proceso.

| | |
|---|---|
| **Nombre** | Christian Enrique Zhiminaicela Segarra |
| **Categoría Profesional** | Estudiante |
| **Rol** | Desarrollador del Backend |
| **Responsabilidad** | Análisis, diseño y programación de la plataforma |
| **Información de contacto** | czhiminaicelas@est.ups.edu.ec |

Tabla 5.2: Personal involucrado

| | |
|---|---|
| **Nombre** | Christian Enrique Zhiminaicela Segarra |
| **Categoría Profesional** | Estudiante |
| **Rol** | Desarrollador del Frontend |
| **Responsabilidad** | Análisis, diseño y programación de la plataforma |
| **Información de contacto** | czhiminaicelas@est.ups.edu.ec |

Tabla 5.3: Personal involucrado

| | |
|---|---|
| **Nombre** | Gabriel Alejandro León Paredes |
| **Categoría Profesional** | Administrador del proyecto |
| **Rol** | Ingeniero de Sistemas |
| **Responsabilidad** | Verificar y aprobar los sprints del proyecto |
| **Información de contacto** | gleon@ups.edu.ec |

Tabla 5.4: Personal involucrado

| | |
|---|---|
| **Nombre** | Juan Inga Ortega |
| **Categoría Profesional** | Administrador del proyecto |
| **Rol** | Ingeniero Electrónico |
| **Responsabilidad** | Verificar y aprobar los sprint del proyecto |
| **Información de contacto** | jinga@ups.edu.ec |

Tabla 5.5: Personal involucrado

### 5.2.3 Requerimientos Funcionales

Los requerimientos funcionales según [37] "describen lo que el sistema debe hacer", estos plantean las características que el sistema tiene que adoptar con respecto a cómo el usuario final interactuara con él. Para lograr la obtención de estos, se ha formulado los siguientes mecanismos detallados a continuación para consolidar con el usuario la información necesaria antes de construir el sistema.

| | |
|---|---|
| **Código** | RE01 |
| **Nombre** | Autenticación del Usuario |
| **Descripción** | La aplicación permitirá al usuario hacer un inicio de sesión obligatorio para poder acceder a sus funcionalidades. |
| **Entradas** | Usuario, Contraseña |
| **Proceso** | Para el acceso del usuario a la aplicación, contara con un formulario donde se tiene que rellenar con los datos que se piden, luego existirá una validación de los datos para ver si son correctos, en caso de ser correctos procederá a seguir a la siguiente interfaz, dependiendo del rol de cada usuario, caso contrario se volverán a pedir rellenar los campos. |
| **Salidas** | Mensaje de error al no ser rellenado los datos correctamente. |
| **Prioridad** | Alta |

Tabla 5.6: Requisito Funcional 1 (Elaboración propia)

| | |
|---|---|
| **Código** | RE02 |
| **Nombre** | Registro de Nuevo Usuario |
| **Descripción** | Registra los datos de un nuevo usuario dentro de Framework Spring Boot |
| **Entradas** | Nombre, Apellido, Nombre de Usuario, Rol, Correo Electrónico, Contraseña |
| **Proceso** | Para registrar al usuario se deberá de contar con un botón que permita registrar el nuevo usuario, el mismo deberá de contar con un formulario en donde se pueda ingresar los datos requeridos. El framework de Spring Boot validara y registrara los datos del nuevo usuario en la base de datos. |
| **Salidas** | Mensaje de registro exitoso de los datos de Usuario. Error al no llenar bien los campos del Formulario |
| **Prioridad** | Alta |

Tabla 5.7: Requisito Funcional 2 (Elaboración propia)

| | |
|---|---|
| **Código** | RE03 |
| **Nombre** | Almacenamiento en BD POSTGRES |
| **Descripción** | Almacenar toda la información de los nuevos usuarios dentro de la base de datos PostgreSQL |
| **Entradas** | Usuarios, roles |
| **Proceso** | Dentro de una base de datos se registrará la información personal y el rol que tendrá cada nuevo usuario registrado, que serán creados por un administrador principal, dentro de la aplicación se validara que los datos ingresados sean correctos para su registro en la base, caso contrario no se registra hasta ingresar datos válidos. |
| **Prioridad** | Alta |

Tabla 5.8: Requisito Funcional 3 (Elaboración propia)

| | |
|---|---|
| **Código** | RE04 |
| **Nombre** | Gestionar Usuarios |
| **Descripción** | Listar, actualizar y eliminar todos los usuarios que estén ingresados dentro de la aplicación. |
| **Entradas** | Usuario, Contraseña, Roles |
| **Proceso** | Para la gestión de los usuarios, solo el administrador de la aplicación tendrá acceso a esa funcionalidad, podrá visualizar todos los usuarios que existen dentro de la base de datos, también puede actualizar y eliminar dichos usuarios. |
| **Prioridad** | Alta |

Tabla 5.9: Requisito Funcional 4 (Elaboración propia)

| | |
|---|---|
| **Código** | RE05 |
| **Nombre** | Almacenamiento de la DATA en MYSLQ |
| **Descripción** | Almacenar información que envié el servidor ChirpStack |
| **Entradas** | EndPoint, puerto |
| **Proceso** | Para este requerimiento se requiere establecer una conexión de ChirpStack con el backend mediante el protocolo http, para recibir y almacenar la DATA que generan los nodos conectados al ChirpStack. |
| **Salidas** | Almacenar dentro de la Base de Datos MySQL la data que envían los nodos conectados al servidor ChirpStack. |
| **Prioridad** | Alta |

Tabla 5.10: Requisito Funcional 5 (Elaboración propia)

| | |
|---|---|
| **Código** | RE06 |
| **Nombre** | Mostrar DATA de los nodos mediante gráficas |
| **Descripción** | Visualizar las gráficas mediante el FrontEnd de la data almacenada de cada nodo. |
| **Entradas** | Nodo, Fecha, Tiempo |
| **Proceso** | Mediante la selección del nodo, fecha y tiempo, el FrontEnd visualizara la gráfica sobre la data almacenada dentro de la base de datos, la misma que envían los nodos desde el servidor ChirpSatck. |
| **Salidas** | Visualización de la gráfica de cada nodo |
| **Prioridad** | Alta |

Tabla 5.11: Requisito Funcional 6 (Elaboración propia)

### 5.2.4 Requerimientos no Funcionales

| | |
|---|---|
| **Código** | RENF01 |
| **Nombre** | Integridad de datos |
| **Descripción** | Cifrar la información de ingreso al sistema. |
| **Entradas** | Datos Usuario, Contraseña |
| **Proceso** | Mediante JWT y OAuth2 manejamos la seguridad, realizando encriptación de la contraseña y manejando los tokens para proporcionar acceso autorizado a los recursos del sistema. |
| **Prioridad** | Alta |

Tabla 5.12: Requisito No Funcional 1 (Elaboración propia)

### 5.2.5 Disponibilidad

El sistema tendrá una disponibilidad de 24 horas, todos los días, es decir, todo el tiempo y así garantizar que el sistema funcione correctamente, ya que este sistema IoT se encontrara instalado en el servidor del grupo de investigación GIHP4C.

### 5.2.6 Seguridad

Garantizar la seguridad de la información, obtenido por los nodos que se encuentran instalados dentro de la Universidad Politécnica Salesiana, en donde la información generada se podrá visualizar solo por el personal a cargo o por usuarios administrativos.

## 5.3 Diagramas para la Aplicación Web

### 5.3.1 Historias de Usuario

| **Código**<br>HU-1 | **USUARIO**<br>Usuario Final |
|---|---|
| **Nombre de Historia** | Registro de Usuarios. |
| **Prioridad**<br>Alta | **Riesgo en Desarrollo**<br>Baja |
| **Desarrollador Responsable** | Christian Zhiminaicela. |
| **Descripción** | Yo como administrador Quiero registrar usuarios para que puedan ingresar a la plataforma web. |
| **Validación** | Debo poder registrar usuarios utilizando datos, como el nombre, el apellido, nombre de usuario, correo electrónico y la contraseña. |

Tabla 5.13: Historia de Usuario 1.

| **Código**<br>HU-2 | **USUARIO**<br>Usuario Final |
|---|---|
| **Nombre de Historia** | Inicio de Sesión. |
| **Prioridad**<br>Alta | **Riesgo en Desarrollo**<br>Baja |
| **Desarrollador Responsable** | Christian Zhiminaicela. |
| **Descripción** | Yo Como usuario quiero iniciar sesión mediante el uso de credenciales para poder acceder a la plataforma web y visualizar la información de acuerdo con el rol de cada usuario. |
| **Validación** | Se validará que cada usuario pueda ingresar a la plataforma mediante su usuario y clave de acuerdo con el rol que tenga. (Administrado o Usuario Normal). |

Tabla 5.14: Historia de Usuario 2.

| **Código**<br>HU-3 | **USUARIO**<br>Usuario Final |
|---|---|
| **Nombre de Historia** | Visualizar la Data. |
| **Prioridad**<br>Alta | **Riesgo en Desarrollo**<br>Baja |
| **Desarrollador Responsable** | Christian Zhiminaicela. |
| **Descripción** | Yo como administrador quiero visualizar la data almacenada en la base de datos de cada nodo que se encuentra conectado al servidor ChirpSatck mediante gráficas. |
| **Validación** | Se validará que la plataforma IoT nos permita visualizar la data mediante gráficas lineales de acuerdo con el nombre del nodo, por fechas, horas, días, semanas y meses. |

Tabla 5.15: Historia de Usuario 3.

| **Código**<br>HU-4 | **USUARIO**<br>Usuario Final |
|---|---|
| **Nombre de Historia** | Gestionar Usuarios |
| **Prioridad**<br>Alta | **Riesgo en Desarrollo**<br>Baja |
| **Desarrollador Responsable** | Christian Zhiminaicela. |
| **Descripción** | Yo como usuario administrativo quiero crear nuevos usuarios ya sean administrativos o normales, y poder visualizar la información de cada usuario existente para poder gestionarlo. |
| **Validación** | Debo poder gestionar la información de cada usuarios existentes de acuerdo a mi rol administrativo. |

Tabla 5.16: Historia de Usuario 4.

| **Código**<br>HU-5 | **USUARIO**<br>Usuario Final |
|---|---|
| **Nombre de Historia** | Cerrar Sesión |
| **Prioridad**<br>Alta | **Riesgo en Desarrollo**<br>Baja |
| **Desarrollador Responsable** | Christian Zhiminaicela. |
| **Descripción** | Yo Como un usuario final quiero cerrar la sesión del sistema para tener un mejor control sobre mi actividad dentro de la plataforma. |
| **Validación** | Debo poder cerrar la sesión, mediante un botón en la parte superior de plataforma. |

Tabla 5.17: Historia de Usuario 5.

### 5.3.2 Prototipos

Previamente se han construido prototipos de pantallas en que las que el usuario final apreciara las diferentes funcionalidades y características que el sistema tendrá cuando ya se encuentre finalmente construido.

**Ingresos a la Plataforma**

Mediante esta pantalla login podrán ingresar al sistema IoT, solo los usuarios registrados, ya sean usuarios administradores o usuarios normales.


Figura 5.1: Pantalla Login

**Pantalla Principal de Inicio**

La siguiente pantalla nos mostrará el Inicio de la plataforma IoT, esta pantalla será la misma para todos los usuarios, sin importar su rol.


Figura 5.2: Pantalla de Inicio

**Gestión de Usuarios**

Mediante el menú se puede ingresar a la pantalla Gestión de Usuarios, la cual nos va a ayudar crear, actualizar y eliminar a un usuario, siempre y cuando el usuario que ingreso al sistema sea un administrador.

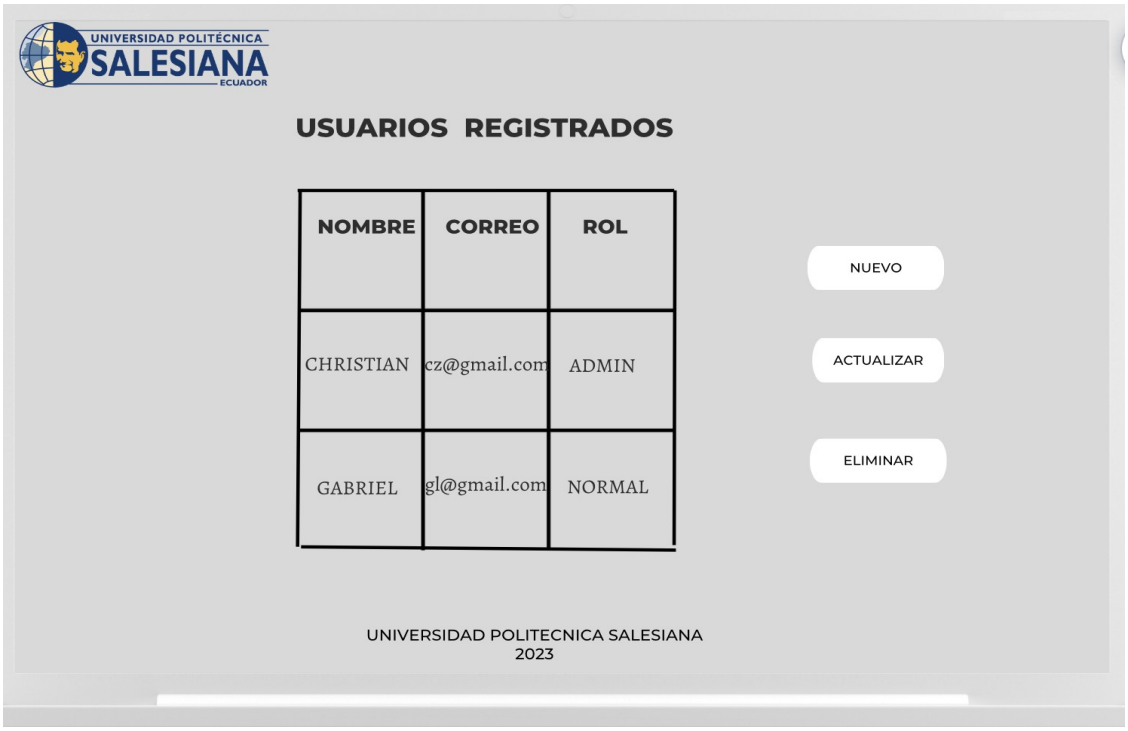



Figura 5.3: Pantalla de Gestión Usuarios

**Registro de Nuevos Usuarios**

En esta pantalla se procede a registrar nuevos usuarios para que tengan acceso al sistema, este proceso solo puede ser realizado por un administrador


Figura 5.4: Pantalla de Registro

**Listar Usuarios Existentes**

En esta pantalla se puede mostrar todos los usuarios existentes con su rol, esta pantalla está habilitada para todos los usuarios sin importar su rol, para que puedan consultar cualquier inconveniente con el administrador.


Figura 5.5: Pantalla de Listado

**Gráficas Lineales**

Esta pantalla nos permite mostrar las gráficas de la data procesada de cada nodo de co2, dando la opción de elegir el nodo y la fecha, para poder visualizar la información, la pantalla está habilitada para todos los usuarios en general.

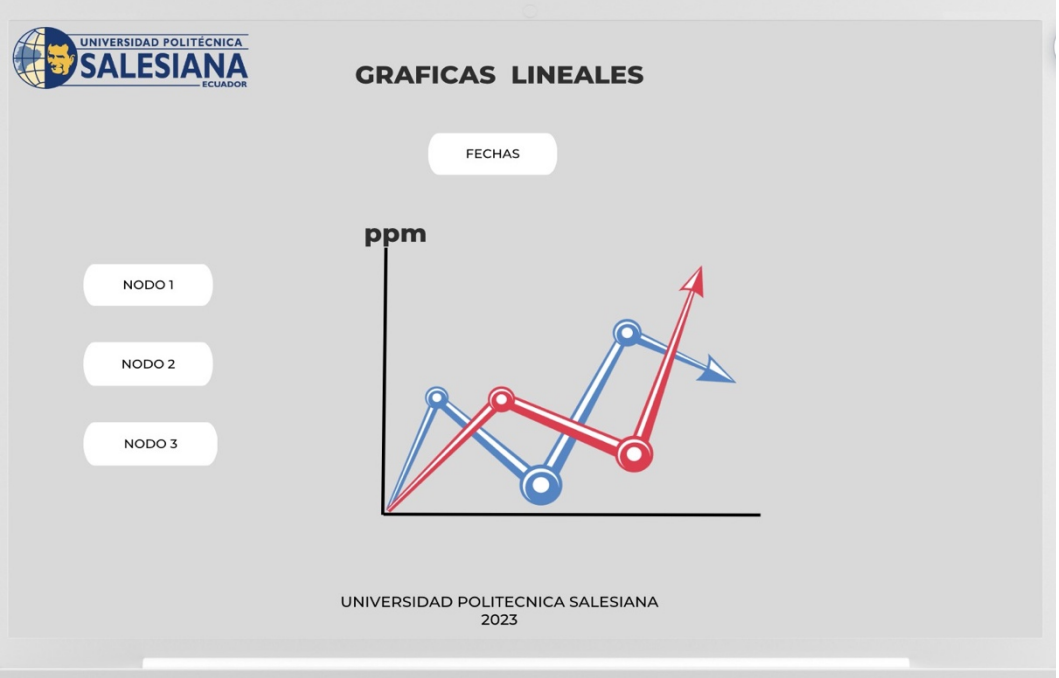



Figura 5.6: Pantalla de Gráficas Lineales

**Gráficas de Nanómetro**

Esta pantalla nos permite mostrar las gráficas de la data procesada de cada nodo de co2, en forma de un nanómetro, teniendo la opción elegir el nodo y la fecha, para poder visualizar la información, la pantalla está habilitada para todos los usuarios en general.


Figura 5.7: Pantalla de Gráficas de Nanómetro

# Capítulo VI

# Arquitectura de Software de Internet de las Cosas

En el siguiente capítulo presentamos la implementación de la arquitectura basada en microservicios para la plataforma de internet de las cosas (IoT) , que se utilizó para la implementación del proyecto.

## 6.1 Diseño de la Arquitectura

La arquitectura del software muestra el funcionamiento del sistema, así como la estructura y la función que tiene que cumplir cada uno de los componentes. El software este compuesto por el backend donde se maneja toda la lógica de negocio, y el frontend donde se visualiza la información mediante una interfaz bien definida para el usuario final. A continuación, se observa el diseño de la arquitectura antes mencionada.

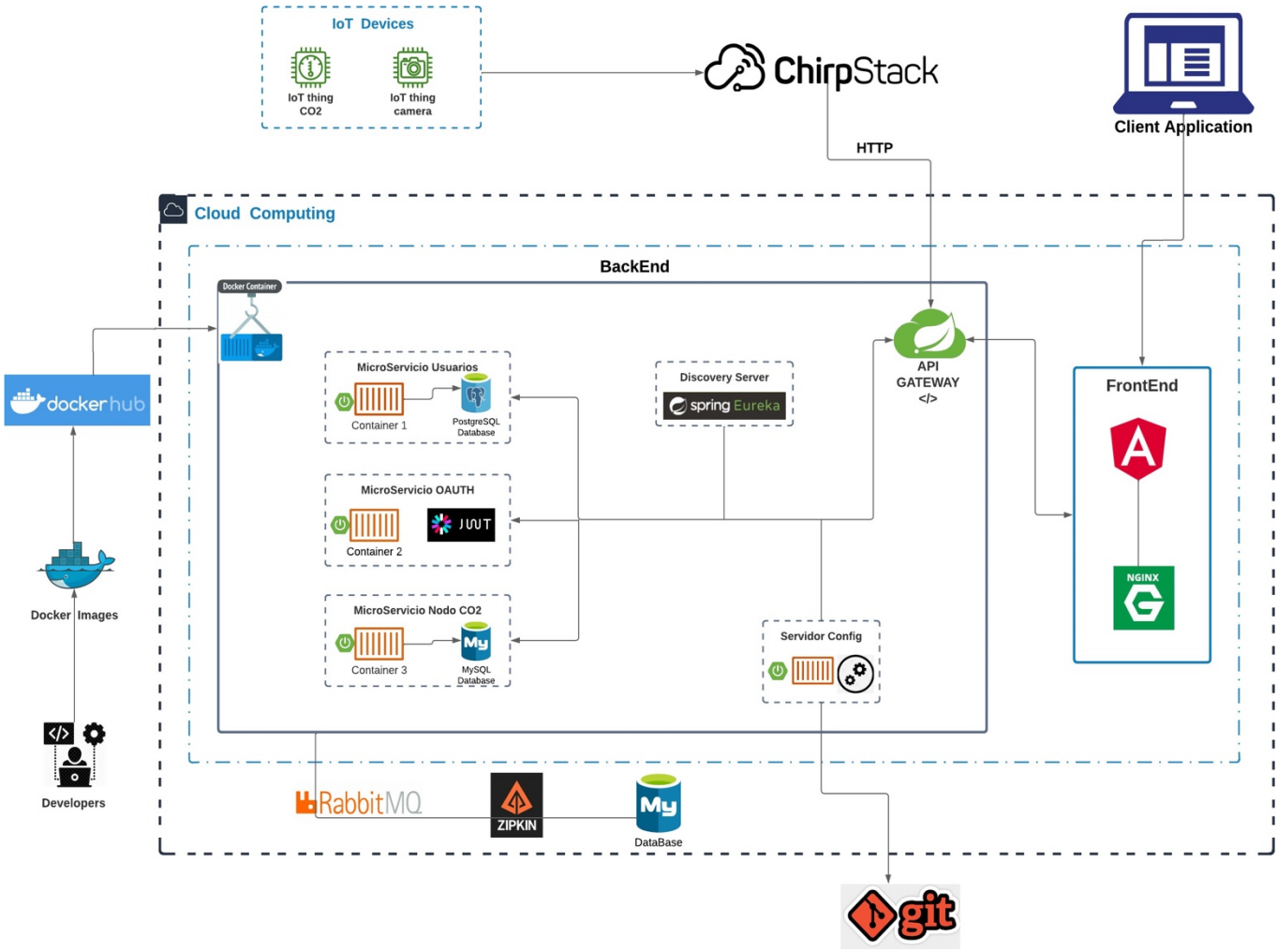



Figura 6.1: Diseño de la Arquitectura para el Software IoT.

## 6.2 Herramientas para el desarrollo del software

El sistema IoT está construido por distintas tecnologías como se puede apreciar en la arquitectura anterior donde cada una de estas cumple una función en especial para el correcto funcionamiento de la plataforma.

A continuación, les mostramos las herramientas tecnológicas utilizadas para el desarrollo del software:

- Spring Boot 2
  - Java Spring Framework Versión: 2.5.3
  - Spring Cloud Security (OAuth2 y JWT "Json Web Token") Version: 2.3.8
  - Spring Cloud Gateway Versión: 3.1.3
  - Spring Cloud LoadBalancer Versión: 3.1.3
  - Spring Cloud Config Server Version: 3.1.3
- Eureka Server Versión: 3.1.3
- Docker Versión: 4.13.1
  - Docker Images
  - Docker Container
  - Docker Hub
- Base de Datos
  - MySQL Versión: 8
  - PostgreSQL Versión: 14
- Postman Versión: 9.31.27
- Zipkin Versión: 2.23.16
- RabbitMQ Versión: 3.10.0
- GitLab Versión: 16.2
- Angular Versión: 12
- ChirpStack
- Protocolo Http

## 6.3 Implementación de la Arquitectura

### 6.3.1 Dispositivos IoT

Estos dispositivos son nodos de sensores de Co2 que están instalados en la Universidad Politécnica Salesiana que se encuentran conectados a una red, estos dispositivos nos permiten la interacción entre el mundo físico y el mundo digital [27]. Tienen la capacidad de transmitir datos en este caso envían la información recolectada de su entorno al servidor chirpStack.

### 6.3.2 ChirpStack

El servidor ChirpStack nos permite administrar los nodos conectados a la red gracias a la interfaz web que posee, mediante su interfaz se puede agrupar los nodos y redireccionar la data que generan hacia nuestro servidor backend, donde vamos a contar con un microservicio que nos permite recibir y al mismo tiempo almacenar la data enviada desde el servidor chirpstack.

**Integración de ChirpStack con el BackEnd**

1. Para la integración les facilitamos el endpoint de la puerta de ingreso "API Gateway" de nuestro backend para que el servidor chirpstack reenvié la data obtenida de los nodos, para este proceso ingresamos a la parte de la integración y seleccionamos el protocolo de comunicación que está recibiendo el backend, en este caso elegimos HTTP como se muestra en la imagen.

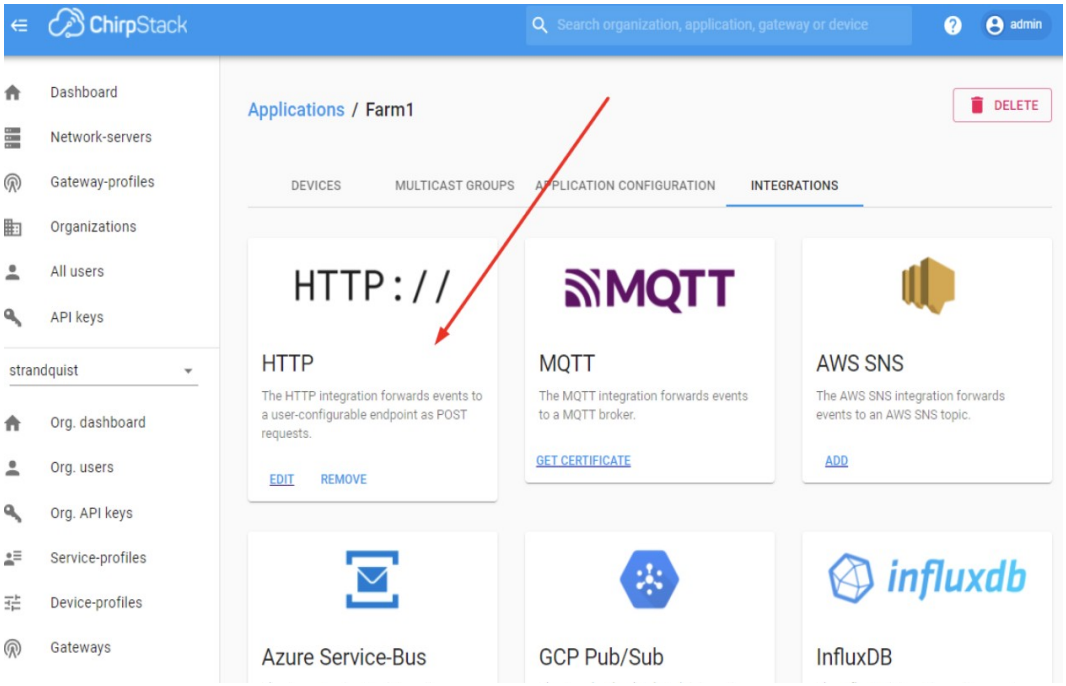



Figura 6.2: Integración HTTP.

2. En esta sección actualizamos la integración mediante el protocolo HTTP (Protocolo de Transferencia de Hipertexto, tal como podemos observar su concepto en el capítulo IV), luego se selecciona el formato json que va a ser decodificado en el payload que envía la data al servidor, luego ingresamos el endpoint con el puerto especifico de nuestro backend el cual está listo para recibir y almacenar la data. El servidor chirpstack realiza una petición POST hacia el backend, nuestro servidor debe aceptar esa petición POST y realizar el proceso de almacenar la data en una base de datos. Finalmente damos clic en actualizar integración y se guardan los cambios realizados y empieza el proceso de envió de la data generada por los nodos.

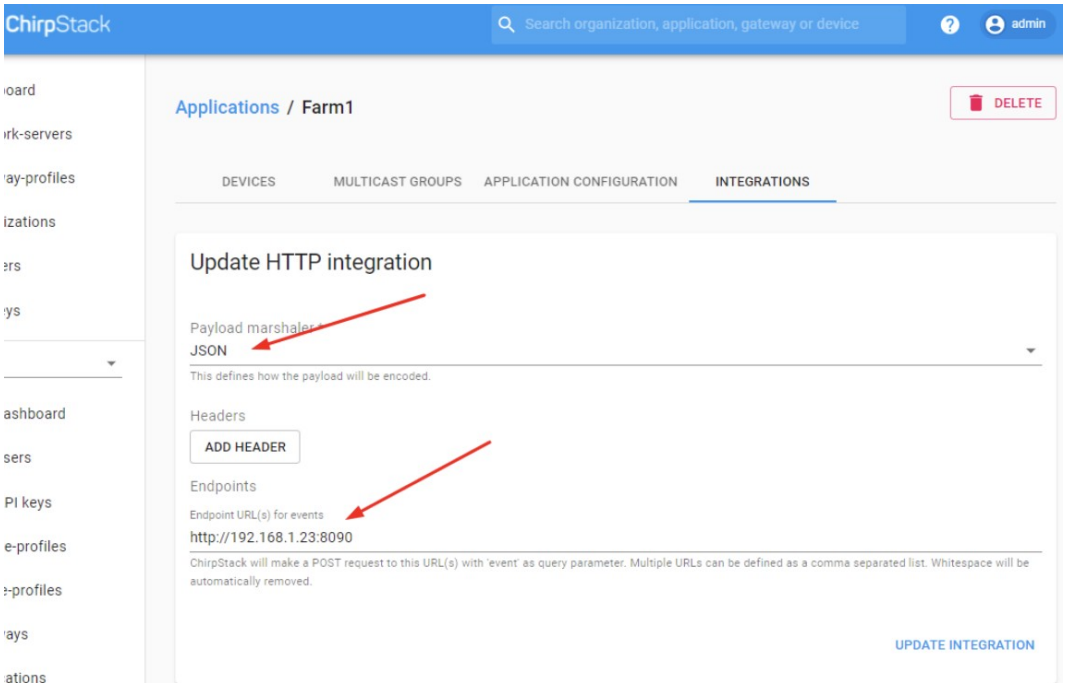



Figura 6.3: Configuración del protocolo HTTP.

### 6.3.3 Backend basado en Microservicios

Para el desarrollo del software IoT, primero empezamos desarrollando la parte del backend, el cual está conformado de varios microservicios para el correcto funcionamiento. Para el desarrollo del backend se utilizó la herramienta tecnología Spring Boot. A continuación, describimos la implementación de cada microservicio.

**EUREKA SERVER**

Eureka Server es el primer servicio en ser creado, ya que su implementación es necesaria para arquitecturas en microservicios porque permite que cada servicio se registre en el servidor de nombres de eureka. El servidor cuenta con una interfaz de usuario y puntos finales de API HTTP para su funcionalidad.

En la siguiente imagen 6.4 obtenida de [28] se observa el comportamiento de los servicios con eureka, primero el cliente "microservicio" se registra al servidor, luego el servidor realiza la búsqueda y la consulta necesaria para poder conectar con el servicio.

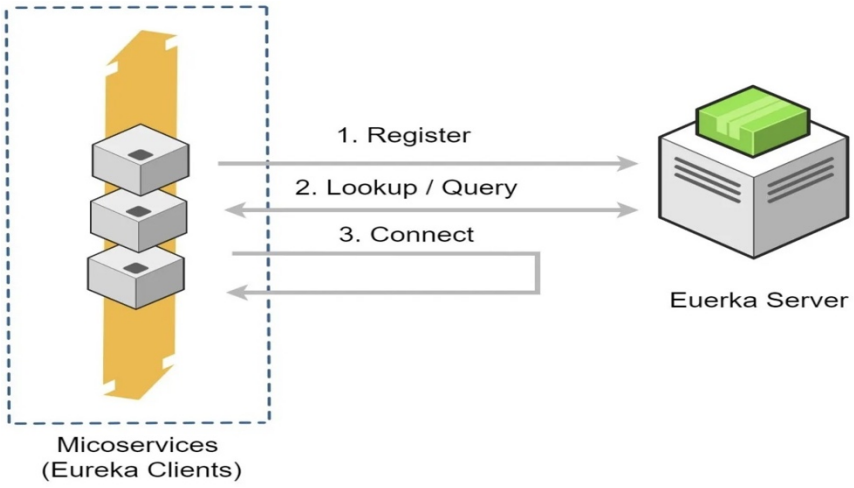



Figura 6.4: Patrón de registro y descubrimiento del servicio. [28]

**Alta disponibilidad con el servidor Eureka**

Todos los clientes activan una llamada REST con el servidor Eureka para auto registrarse en eureka, cuando ya están registrados tienen que enviar latidos del corazón cada 30 segundos para mantener sus registros actualizados y poder saber el estado en el que se encuentra cada servicio, Cuando se produce un apagado y el servidor no recibe una notificación después de 90 segundos los clientes activan otra llamada REST para que el servidor pueda eliminar las instancias del registro, para esta finalidad el servidor cuenta con los siguientes estados:

- OUT OF SERVICE
- DOWN
- STARTING
- UNKNOWN
- UP

Los servicios clientes cuentan con una memoria caché para almacenar los registros de eureka, por lo que no tienen que solicitar al servidor cada vez que necesite llamar a un servicio. Cada cliente de Eureka requiere al menos una URL de servicio para localizar a un par. Si no se llena sus registros con mucho ruido.

**Desplegando el Servidor Eureka**

Primero creamos el proyecto basado en spring boot y procedemos a la configuración del servicio, es necesario agregar la siguiente dependencia.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Figura 6.5: Dependencia Netflix Eureka Server.

Después de agregar la dependencia necesaria, continuamos declarando el servicio con la anotación @EnableEurekaServer como se puede apreciar la siguiente imagen:

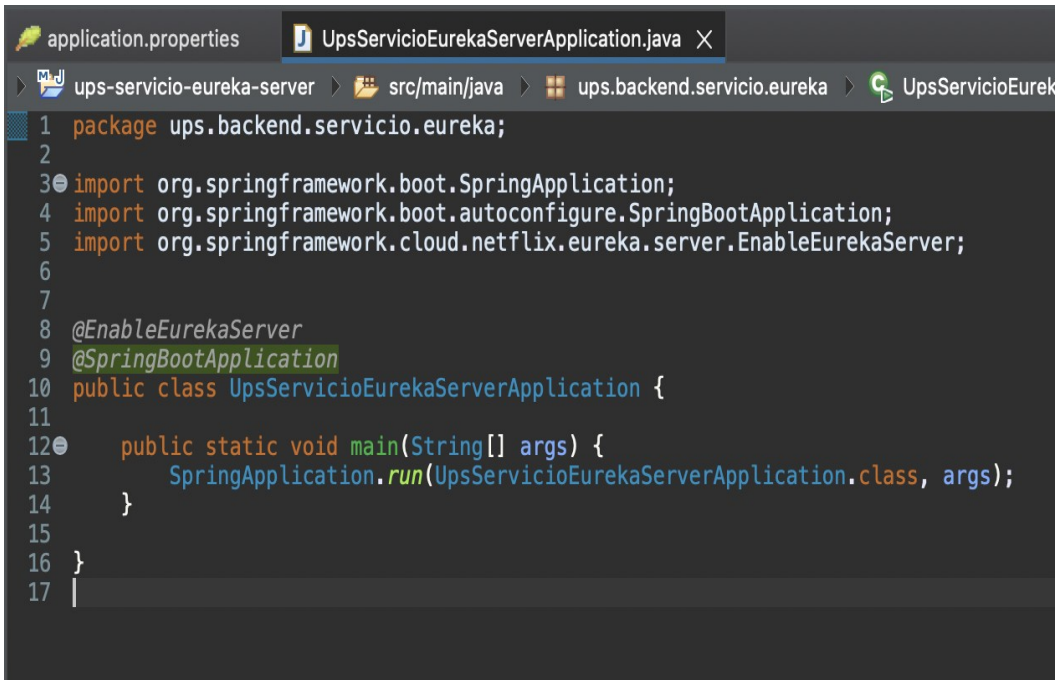



Figura 6.6: Anotación Eureka Server.

Configuración del archivo application.properties para el servicio eureka, declaramos el nombre del servicio y el puerto que va a utilizar, así como se muestra en la siguiente imagen.

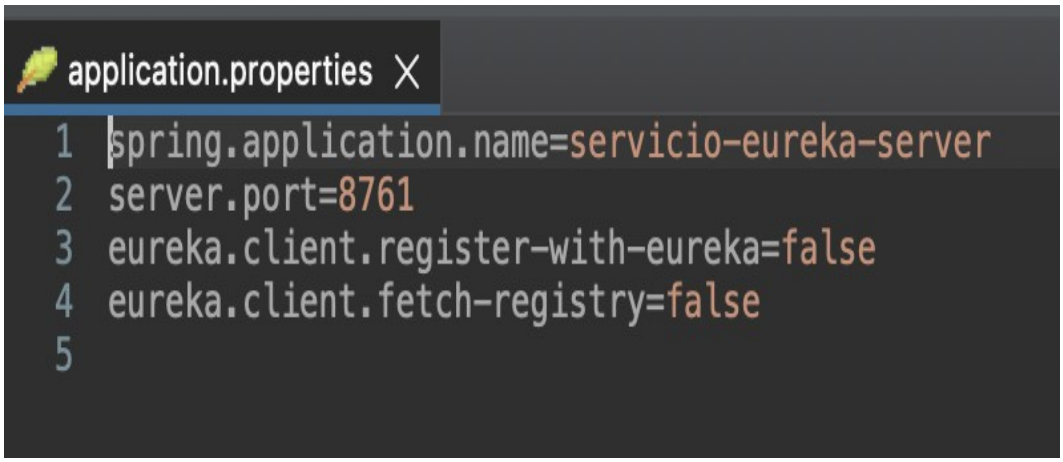



Figura 6.7: Archivo de configuración eureka.

## MICROSERVICIO NODOCO2

Microservicio CO2 se desarrolló para recibir la data de los nodos de Co2 que son enviados desde el servidor chirpstack mediante el endpoint configurado, y guardar la data en una base de datos MySQL. La implementación del microservicio fue basada en la siguiente arquitectura de 3 capas:

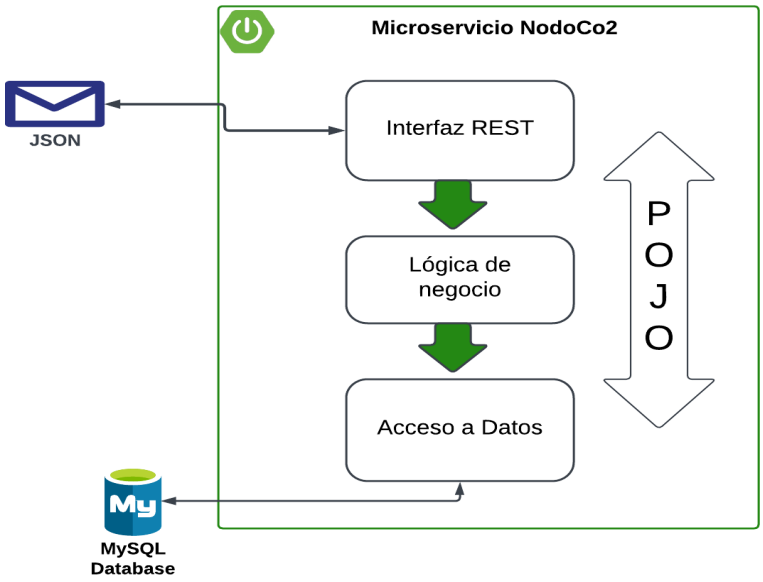



Figura 6.8: Arquitectura 3 capas para NodoCo2.

### Acceso a Datos con JpaRepository para NodoCo2

Spring Boot nos brinda una forma rápida y segura de crear la clase de acceso a datos, es implementando la interfaz JpaRepository de Spring Data JPA, para utilizar esta interfaz tenemos que agregar la dependencia de **JPA** y la dependencia para la base de datos **MYSQL**, dentro del archivo **POM.XML**. Como otro punto es la creación de la entidad que representara nuestra tabla en una base de datos, para este proceso tenemos que utilizar la anotación **@Entity** de JPA.

Una vez que declaremos nuestra entidad llamada "SensorcO2", procedemos a crear nuestro repositorio basado en Springframework, que nos permite utilizar la interfaz de JpaRepository para poder utilizar todos los métodos internos, la clase se crea de la siguiente manera:

**public interface SensorCo2Repository extends JpaRepository<SensorCO2, Long> { }**

**Lógica de negocio para la data enviada desde chirpstack**

En esta clase es donde tenemos que manejar las operaciones, reglas, y toda la lógica de negocio que necesite nuestro sistema. Creamos una clase llamada "SensorCo2Service" como proveedor de servicios, mediante la cual se accede al JpaRepository y sus métodos, para este proceso necesitamos declarar las siguientes anotaciones:

- @Service: Cree una instancia "bean" de esta clase que nos permitirá usar en otras instancias.
- @Autowired: Permite utilizar la instancia "bean" a la variable que tiene la anotación @Autowired, en este caso la variable es la interfaz creada anteriormente en la capa de acceso a datos "SensorCo2Repository"

En la siguiente imagen se visualiza estas anotaciones declaradas anteriormente.

```java
@Service
public class SensorCo2Service {

    @Autowired
    SensorCo2Repository sensorCo2Repository;


    public RequestMessage registrarMedicionCo2(Root root) {
        JSONObject json = new JSONObject(root.getObjectJSON());

        SensorCO2 sensorCO2 = new SensorCO2();

        sensorCO2.setFecha(root.getRxInfo().get(0).getTime());
        sensorCO2.setFechaBusq(root.getRxInfo().get(0).getTime());
        sensorCO2.setHora(root.getRxInfo().get(0).getTime());
        sensorCO2.setDeviceName(root.getDeviceName());
        sensorCO2.setCo2_A(json.getDouble("Co2_A"));
        sensorCO2.setCo2_b(json.getDouble("Co2_b"));

        RequestMessage rm = new RequestMessage();

        try {
            sensorCo2Repository.save(sensorCO2);
            rm.setCode("0");
            rm.setMessage("Peticion ingresada correctamente: " + json.toString());
        } catch (Exception e) {
            rm.setCode("1");
            rm.setMessage("Fallo WS" + e.getMessage());
        }
        return rm;
    }
```

Figura 6.9: Clase SensorCo2Service.

Dentro de la clase SensorCo2Service tenemos un método que nos permite recibir el json enviado desde chirpstack, tal como se observa en la imagen 6.9, para este proceso importamos la siguiente librería dentro de la clase:

**import org.json.JSONObject;**

Para poder utilizar la librería mencionada, agregamos en el archivo pom.xml la siguiente dependencia **org.json** la versión **20201115**.

El método procesa el json, recibiendo como atributo un POJO, llamado "Root" el cual contiene todas las variables del json recibido, y la otra clase es la entidad "SensorCO2" que está compuesta de los atributos que vamos a necesitar para guardar la información necesaria en la base de datos.

**Implementación del Controlador para Servicios Web RestFul**

Spring Boot nos permite crear servicios web RestFul para todo tipo de aplicaciones empresariales, para que spring declare que una clase es un servicio web tenemos que agregar en la clase la anotación @RestController y @RequestMapping("/co2"), el cual nos permite poner un nombre para identificar y realizar peticiones al servicio web.

- **@Autowired:** Esta anotación inyecta una instancia de nuestra clase de lógica de negocio.
- **@PostMapping(path="save"):** Esta anotación declara el tipo de petición y el nombre por el cual va a ser solicitado el método.
- **@RequestBody:** Spring de serializa automáticamente el JSON en un objeto Java, asignado el cuerpo HttpRequest al objeto declarado, en este caso al objeto Root.

En la siguiente imagen se puede apreciar el servicio web con el método POST que esta implementado con un atributo @RequestBody que recibirá el json y procederá a guardar la data, mediante la instancia de lógica de negocio.

```
@RestController
@RequestMapping("/co2")
public class Co2Controller {

    @Autowired
    SensorCo2Service sensorCo2Service;

    @PostMapping(path = "/save")
    public RequestMessage registrar(@RequestBody Root root) {
        return this.sensorCo2Service.registrarMedicionCo2(root);
    }
```

Figura 6.10: Servicio Web RestFul.

**Registrando el microservicio NodoCo2 en Eureka**

Una vez creado el proyecto con spring boot agregamos la dependencia como cliente para conectar con el servidor eureka y agregamos la anotación @EnableEurekaClient en la clase principal.

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Figura 6.11: Dependencia Netflix Eureka Client.

El microservicio NodoCo2 se configuro con un puerto dinámico para poder levantar más instancias del mismo servicio, pero con nuevos puertos, para registrar el servicio a eureka se configura el siguiente archivo properties.

*application.properties

```
spring.application.name=servicio-nodoco2
server.port=${PORT:0}
#server.port=64167

eureka.instance.instance-id=${spring.application.name}:${spring.application.instance_id:${random.value}}

eureka.client.service-url.defaultZone=http://servicio-eureka-server:8761/eureka

spring.sleuth.sampler.probability=1.0

```

Figura 6.12: Archivo de configuración CO2.

**MICROSERVICIO USUARIOS**

El Microservicio Usuarios nos permite la gestión de todos los nuevos usuarios que van a tener acceso a la plataforma, para su desarrollo implementamos interfaz JpaRepository de Spring Data JPA para procesar datos y persistir en la base de datos (PostgreSQL). Para poder utilizar la interfaz tenemos que agregar dentro del archivo pom.xml la dependencia para JPA y para la base de datos postgres.

**Registrando el microservicio usuarios en Eureka Server**

De igual manera como es un servicio que se va a registrar como cliente en eureka server, tenemos que agregar la dependencia correspondiente y en la clase principal agregar la anotación @EnableEurekaClient, en la siguiente imagen se observa la dependencia agregada.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Figura 6.13: Dependencia Netflix Eureka Client.

El servicio se configuro con un puerto dinámico para poder levantar más instancias del mismo servicio, pero con nuevos puertos, para registrar el servicio a eureka se configura el siguiente archivo properties.

application.properties

```
spring.application.name=servicio-usuarios
server.port=${PORT:0}

eureka.instance.instance-id=${spring.application.name}:${spring.application.instance_id:${random.value}}

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

spring.config.import=optional:configserver:

spring.sleuth.sampler.probability=1.0
```

Figura 6.14: Archivo de configuración Usuarios.

**MICROSERVICIO OAUTH**

Este servicio OAUTH está enfocado en la seguridad del sistema para la autenticación de usuarios mediante tokens con JWT, permite proteger y dar acceso a los recursos. El servicio permite generar tokens de acuerdo con el usuario y el rol que tenga cada uno, genera un token que tiene que ser validado por el servidor de autorización para tener acceso a los recursos.

Para el desarrollo de este microservicio se necesita las siguientes dependencias que se visualizan en la imagen:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
    <version>2.3.8.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
    <version>1.1.1.RELEASE</version>
</dependency>
```

Figura 6.15: Dependencias para el servicio OAUTH.

**Spring Security**

Es un framework de seguridad para aplicaciones Java EE que se integra con spring boot, nos permite el manejo de los componentes para autenticación y autorización basándose en credenciales tales como, nombre de usuario y clave. En la siguiente imagen se observa cómo se genera el token de acceso y el permiso a los recursos del sistema.

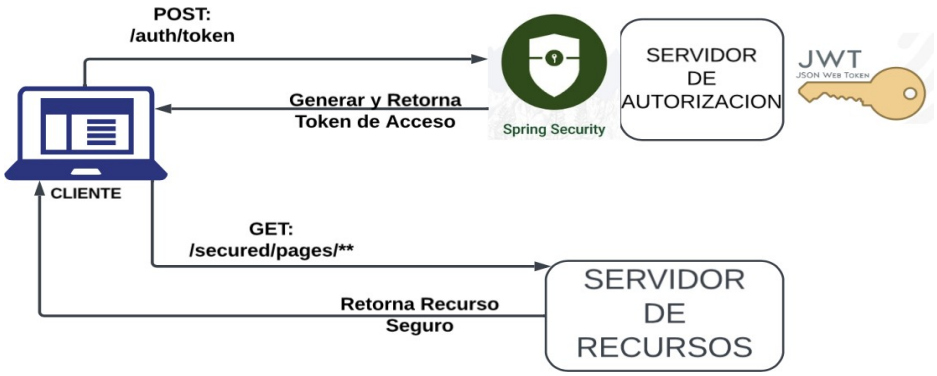



Figura 6.16: Spring Security.

El servidor de autorización es un componente necesario para la seguridad del sistema, actúa como un punto de autorización que permite identificar las características de su aplicación, para poder dar acceso al servidor de recursos, el cual contiene las funcionalidades y recursos necesarios para el funcionamiento del sistema, de acuerdo con el rol de cada usuario que se identifique en el servidor de autorización.

**Registrando el microservicio OAuth en Eureka Server**

En la clase principal agregar la anotación @EnableEurekaClient para que se registre en el servidor de nombres, el servicio se configuro con un puerto estático 9100 dentro del archivo properties, así como se visualiza en la siguiente imagen:

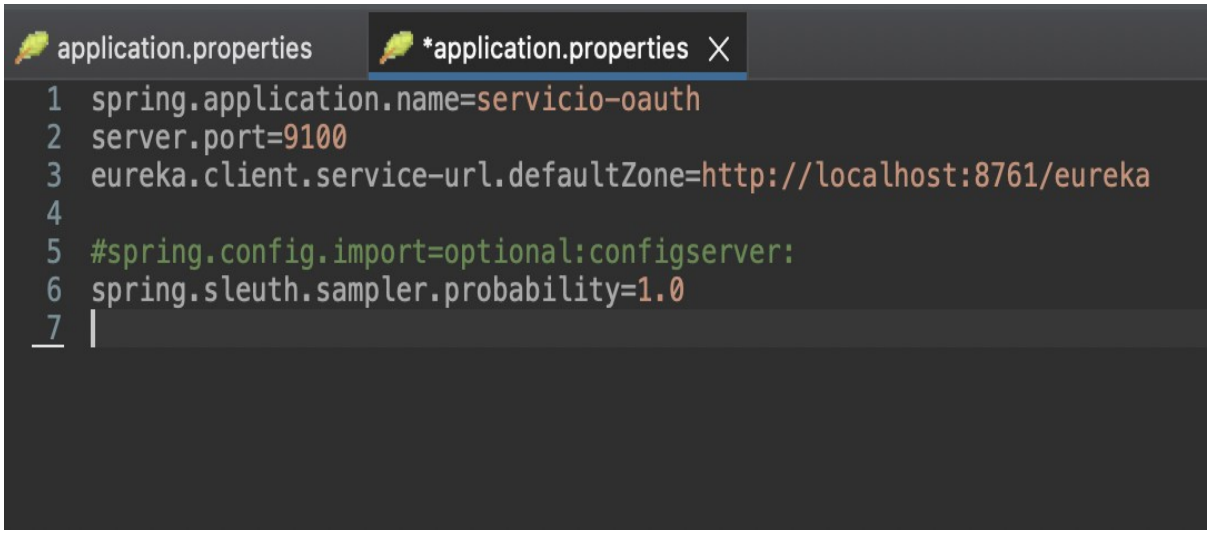



Figura 6.17: Archivo de configuración Oauth.

**SERVICIO API GATEWAY**

Fue desarrollada con Spring Cloud Gateway, la API Gateway es la puerta de entrada principal al backend que permite el acceso a cada microservicio. En la clase principal agregamos la anotación "@EnableEurekaClient" para registrar como cliente en el servidor eureka. Agregamos la siguiente dependencia para poder configurar nuestra Api gateway.

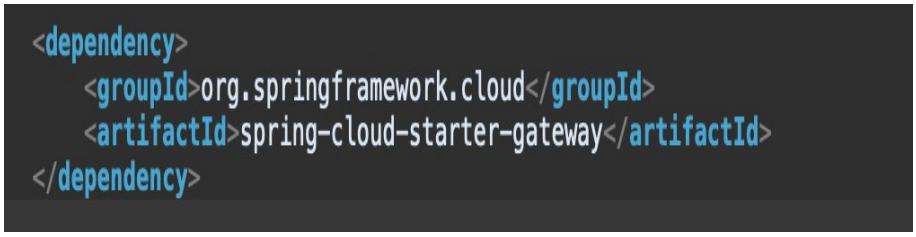



Figura 6.18: Dependencia para la API Gateway.

Para su implementación configuramos el archivo properties con el nombre del servicio y con un puerto estático 8090 que utilizara nuestro gateway. También declaramos la dirección de nuestro servidor eureka para su registro, tal como se aprecia en la imagen siguiente:

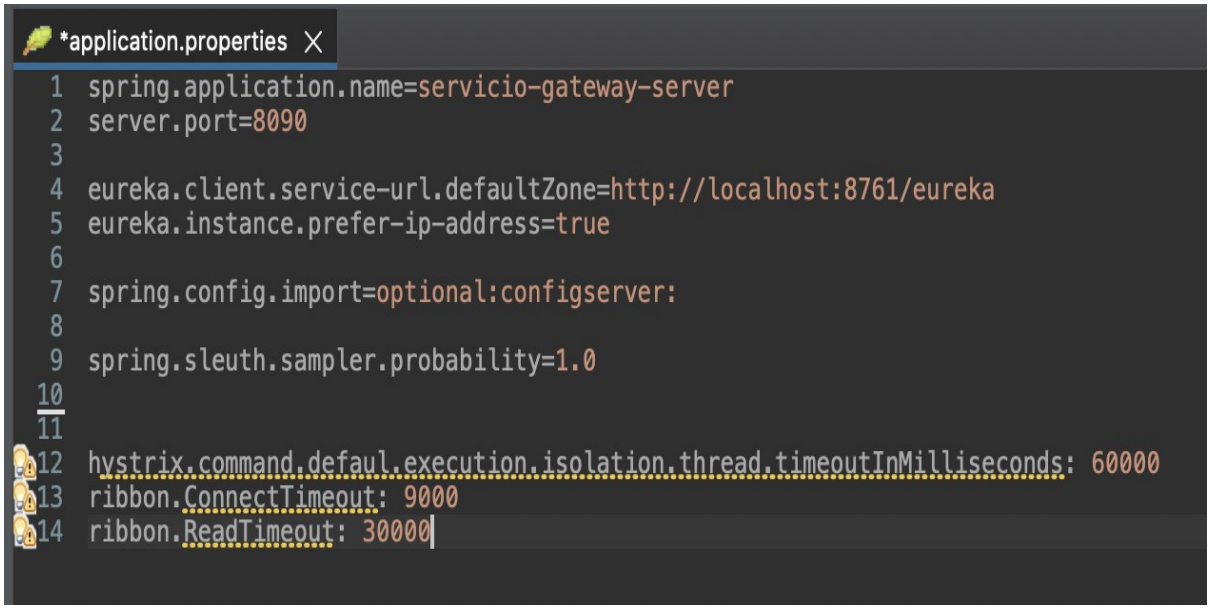



Figura 6.19: Archivo de configuración gateway.

También se configuro las rutas de cada microservicio dentro del archivo application.yml para que la API Gateway nos permita identificar y dar acceso a cada microservicio, dependiendo de la petición que realicen, así protegemos y ocultamos el acceso directo a cada microservicio, dándole seguridad total a los servicios. Toda petición tendrá que ingresar solo por la API principal y se direccionará a los recursos necesarios.

*application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:

      - id: servicio-oauth
        uri: lb://servicio-oauth #lb es balanceo de carga
        predicates:
          - Path=/api/security/**
        filters:
          - StripPrefix=2

      - id: servicio-usuarios
        uri: lb://servicio-usuarios #lb es balanceo de carga
        predicates:
          - Path=/api/usuarios/**
        filters:
          - StripPrefix=2

      - id: servicio-nodoco2
        uri: lb://servicio-nodoco2
        predicates:
          - Path=/api/nodoco2/** #ruta para el  acceso a los endpoints del nodoco2
        filters:
          - StripPrefix=2

```

Figura 6.20: Configuración de las rutas.

**Servidor de Configuración**

Dentro del propio (BanckEnd) se procede a la implementación del servidor de configuración, este servicio nos ayuda a centralizar la configuración de cada microservicio para conectarse a eureka server. Creamos el proyecto con spring boot y procedemos a agregar la siguiente dependencia:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Figura 6.21: Dependencia del servidor de configuración.

Agregamos en la clase principal del servicio la siguiente anotación @EnableConfigServer y @SpringBootApplication para que se establezca como un servidor de configuración para el sistema.

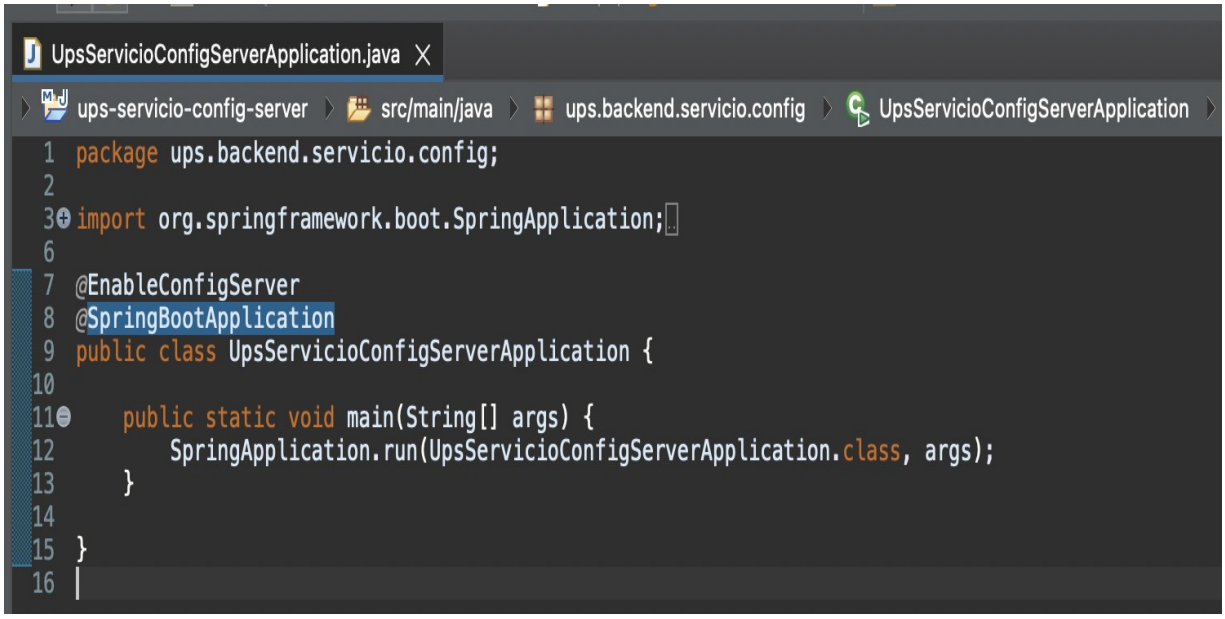



Figura 6.22: Anotaciones del servidor de configuración.

Este microservicio se conecta a gitlab para poder obtener las configuraciones establecidas y que se encarga de distribuir la configuración necesaria a cada servicio, para este proceso se configura el siguiente archivo:

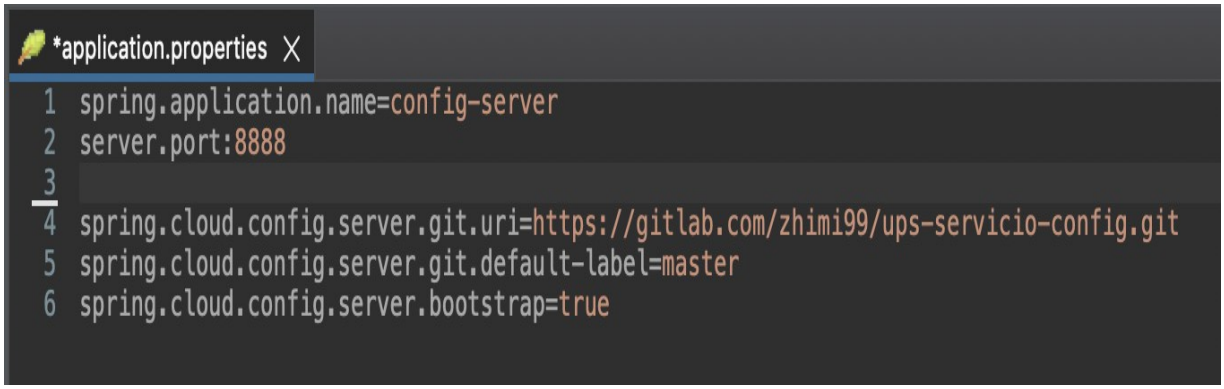



Figura 6.23: Archivo de configuración.

**Centralizando Configuraciones con Gitlab**

Para gestionar las configuraciones de cada servicio por separado es una tarea tediosa y que nos toma mucho tiempo, ya que es necesario realizar los cambios en cada servicio para el

correcto funcionamiento. Por esas razones se implementó un servidor de configuraciones que va a conectar todos los servicios mediante un solo servicio de configuración. Para centralizar todas las configuraciones necesarias, el servidor se conecta a gitlab y dentro de nuestra cuenta en gitlab creamos un repositorio que contenga los archivos necesarios para la configuración de cada servicio. El primer archivo necesario es el **application.properties**, en este archivo agregamos la configuración necesaria para los microservicios, configuración para la seguridad, configuración para conectarse a eureka server y entre otras. Antes de proceder con la centralización tenemos que eliminar la línea que contiene la conexión con eureka **"eureka.client.service-url.defaultzone"** del archivo application.properties de cada microservicio, ya que vamos a centralizar la configuración de eureka solo necesitamos agregar una vez la conexión con eureka dentro del archivo **application.properties** creado en gitlab, a continuación, se puede visualizar en la siguiente imagen dichas configuraciones.

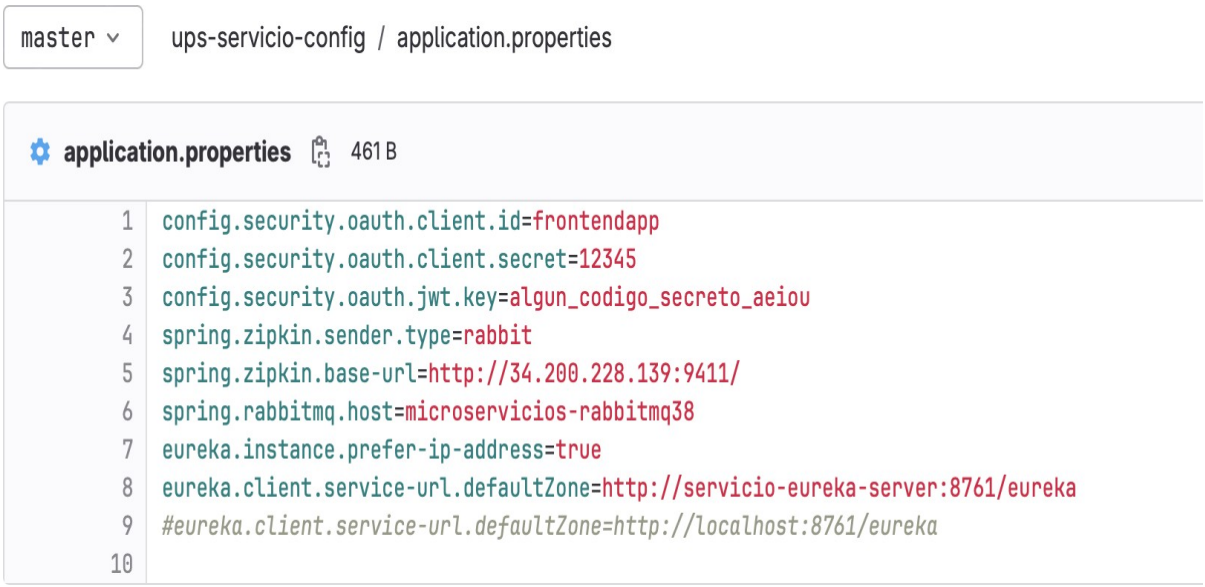



Figura 6.24: Archivo de configuración en Gitlab.

Dentro del repositorio en gitlab creamos otro archivo de configuración para el microservicio usuarios el cual le permite al servicio conectarse con “la base de datos postgres”, donde se va a almacenar toda la información de los usuarios registrados. La siguiente imagen nos muestra la

configuración establecida en el archivo.

master ▾ ups-servicio-config / servicio-usuarios-dev.properties Find file Blame History

servicio-usuarios-dev.properties 460 B Edit ▾ Replace Delete

```
1  spring.datasource.url=jdbc:postgresql://usuariosupsdb.cgvbksvu0sji.us-east-1.rds.amazonaws.com:5432/iotusuarios
2  spring.datasource.username=postgres
3  spring.datasource.password=postgres
4  spring.datasource.driver-class-name=org.postgresql.Driver
5  spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL95Dialect
6  spring.jpa.hibernate.ddl-auto=create
7
8  logging.level.org.hibernate.SQL=debug
9
10 spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
11
```

Figura 6.25: Configuración para PostgreSQL.

También generamos un archivo para las configuraciones del microservicio Co2 que conecta a la base de datos “MySQL” para guardar la data recibida desde el servidor chirpstack.

master ▾ ups-servicio-config / servicio-nodoco2-prod.properties Find f

servicio-nodoco2-prod.properties 443 B Edit ▾

```
1  spring.datasource.url=jdbc:mysql://co2upsdb.cgvbksvu0sji.us-east-1.rds.amazonaws.com:3306/nodoco2?
2  spring.datasource.username=root
3  spring.datasource.password=mysqlups
4  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
5  spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
6  spring.jpa.hibernate.ddl-auto=update
7
8  logging.level.org.hibernate.SQL=debug
9
```

Figura 6.26: Configuración para MySQL.

**Integrando el servidor de configuración en los microservicios**

La conexión del servidor de configuración con gitlab nos permite centralizar las configuraciones para cada microservicio, cada vez que un microservicio se ejecuta primero se conecta al servidor de configuración el cual le da acceso al archivo **application.properties** que esta creado en gitlab para asignar las configuraciones globales como la conexión con eureka server, para realizar la comunicación de los servicios con el servidor de configuración, procedemos a imple- mentar un nuevo archivo **bootstrap.properties** en cada servicio para que nos permita conectar al servidor de configuración el cual conecta a gitlab y establece las configuraciones requeridas en cada microservicio. Las siguientes imágenes nos muestran el archivo bootstrap que contiene la configuración para conectar al servidor de configuraciones y obtener las configuraciones para conectar con eureka server y en si con otros servicios, también nos establece las configuraciones para la conexión a la base de datos especifica de cada servicio, de acuerdo con el perfil que tenga configurado en el archivo bootstrap cada servicio.

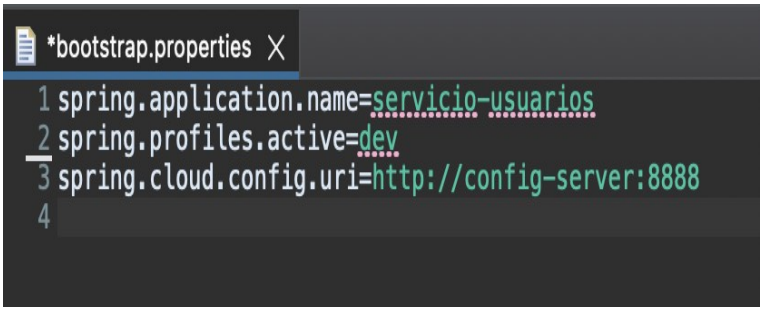



Figura 6.27: Archivo bootstrap de Usuarios.

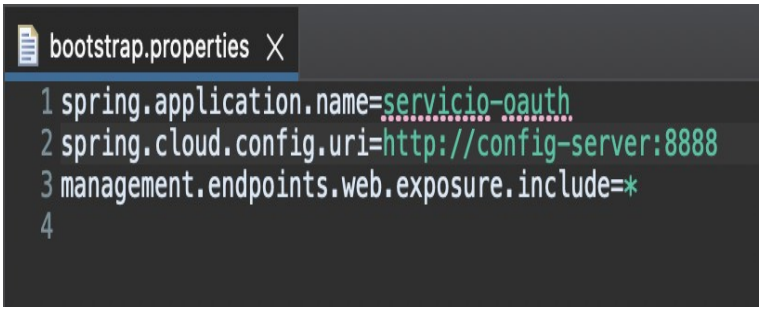



Figura 6.28: Archivo bootstrap de Oauth.

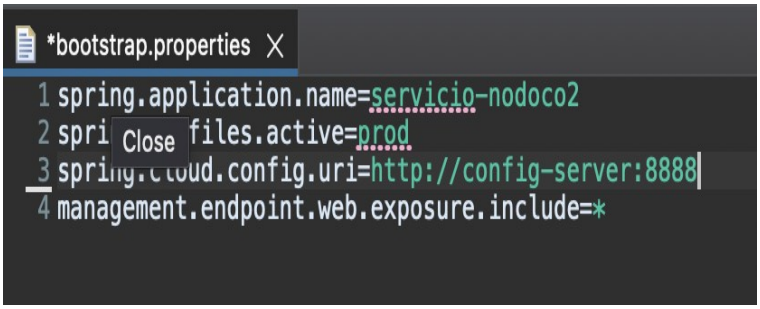



Figura 6.29: Archivo bootstrap de NodoCo2.

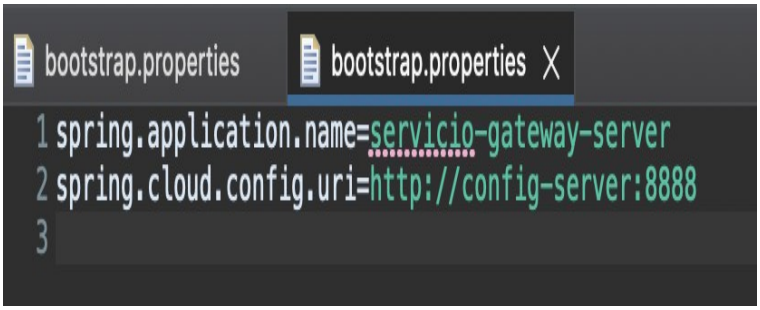



Figura 6.30: Archivo bootstrap del API gateway.

### 6.3.4 Zipkin Server

Dentro de toda esta arquitectura basada en microservicios estamos utilizando una herramienta llamada zipkin server, la cual nos permite rastrear y recopilar datos de las peticiones que se realizan a cada microservicio desde la parte de un cliente, esto nos permite solucionar problemas de latencia en las arquitecturas de microservicio y poder identificar los errores con más facilidad, gracias a las funcionalidades que nos brinda su interfaz se puede visualizar el árbol de llamada de cada traza. El servidor zipkin almacena toda la información de los endpoint de cada servicio, rutas solicitadas.

En el siguiente diagrama 6.31 se observa la implementación del servidor zipkin, configurado para almacenar la información de las trazas en la “base de datos MySQL”.

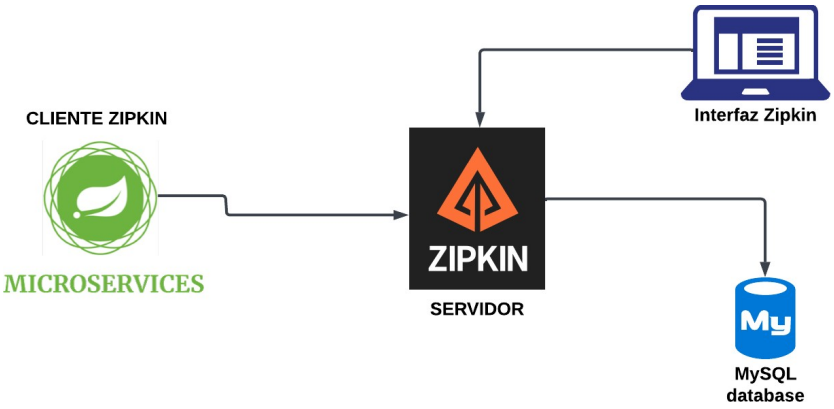



Figura 6.31: Diagrama de la implementación zipkin.

Para integrar con la “base de datos MySQL” y a la vez con un broker de mensajes "RabbitMQ", se configuro el siguiente archivo zipkin:

zipkin

```
#!/bin/sh
export RABBIT_ADDRESSES=localhost:5672
export STORAGE_TYPE=mysql
export MYSQL_USER=zipkin
export MYSQL_PASS=zipkin
export MYSQL_HOST=localhost
export MYSQL_TCP_PORT=3306
java -jar ./zipkin.jar
```

Figura 6.32: Integración de Zipkin con MySQL y Rabbit.

También tenemos que agregar la siguiente línea para completar la integración de zipkin con rabbit, esto se agrega en el archivo **application.properties** creado en gitlab.

**spring.zipkin.sender-type=rabbit**

Cada microservicio se conecta al servidor zipkin mediante su dirección ip que se implementa en el archivo de configuración **application.properties** en gitlab.

**spring.zipkin.base-url="dirección IP del servidor zipkin"**

Se puede apreciar estas configuraciones ya centralizadas en la imagen 6.24 "Archivo de configuración en Gitlab".

### 6.3.5 RabbitMQ

RabbitMQ nos permite manejar un canal básicamente un evento, donde productor y consumidor tienen que suscribirse a este canal, en este caso los microservicios son los productores de mensajes que van a enviar al broker "RabbitMQ" y nuestro consumidor sería zipkin el cual también está suscripto a este canal y recibe cada vez que llega un mensaje, rabbitmq se encarga de este proceso mediante colas de mensaje. En el siguiente diagrama 6.33 se observa la implementación del broker RabbitMQ y como interactúa con el servidor zipkin y sus clientes.

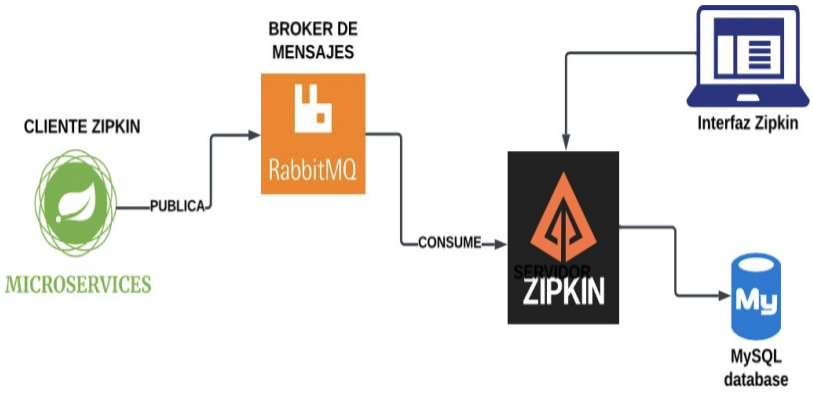



Figura 6.33: Diagrama de la implementación de RabbitMQ.

La conexión de los microservicios con rabbitmq se realiza con la siguiente línea de configuración:

**spring.rabbitmq.host=microservicio-rabbitmq38**

Como ya centralizamos las configuraciones no es necesario agregar en todos los microservicios, solo tenemos que agregar en el archivo de configuración "application.properties" de gitlab tal como se aprecia en la imagen 6.24 "Archivo de configuración en Gitlab".

### 6.3.6 Dockerización de los microservicios

Para poder dockerizar los microservicios, primero tenemos que generar un jar para cada servicio, en la siguiente imagen se puede ver como se genera un jar.

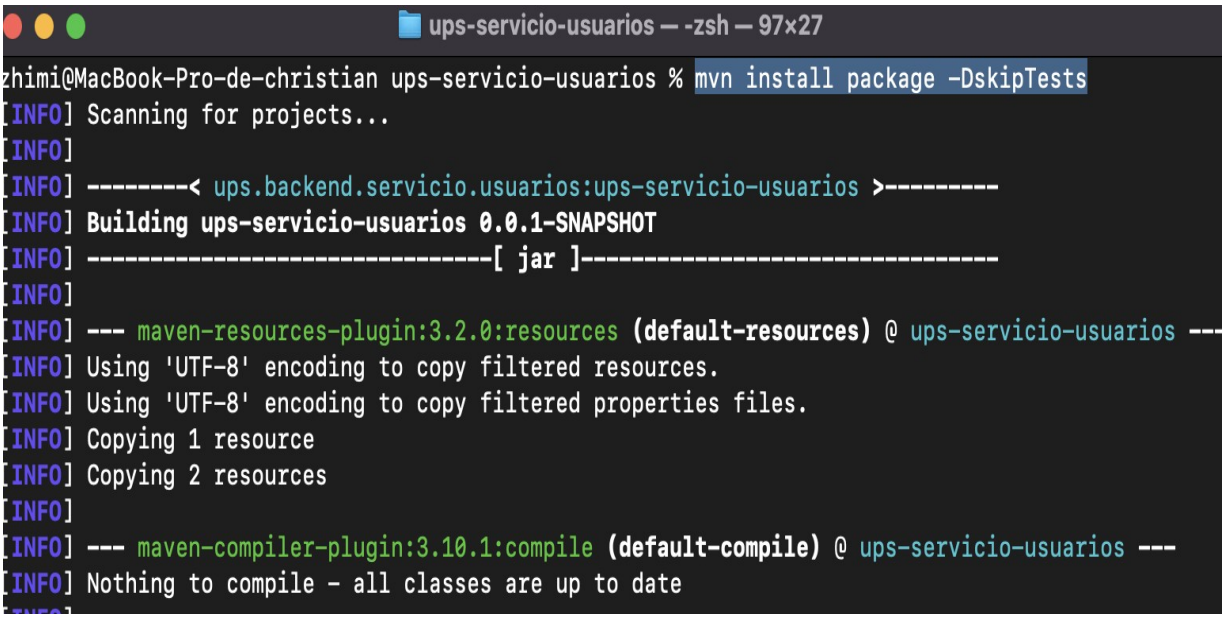



Figura 6.34: Código para generar un jar.

**DockerFile**

Se genero un archivo llamado dockerfile con su respectiva configuración, donde se tiene que agregar el jar generado anteriormente y la versión de java, este proceso se realiza para todos los microservicios, para generar una imagen docker primero tenemos que crear el dockerfile en todos los servicios, en la siguiente imagen se observa el dockerfile creado para el servicio usuario:


Figura 6.35: Archivo dockerfile.

### Docker Imagen

Para generar las imágenes docker debemos tener en cuenta el nombre del servicio y la versión, la siguiente figura nos muestra el código para generar la imagen docker para el servicio usuario, de acuerdo con los cambios que se van realizando se va cambiando la versión de las nuevas imágenes generadas con el mismo nombre, pero con diferente número de versión. Este pro- ceso es el mismo para todos los servicios, solo tenemos que cambiar el nombre del servicio y establecer la versión.

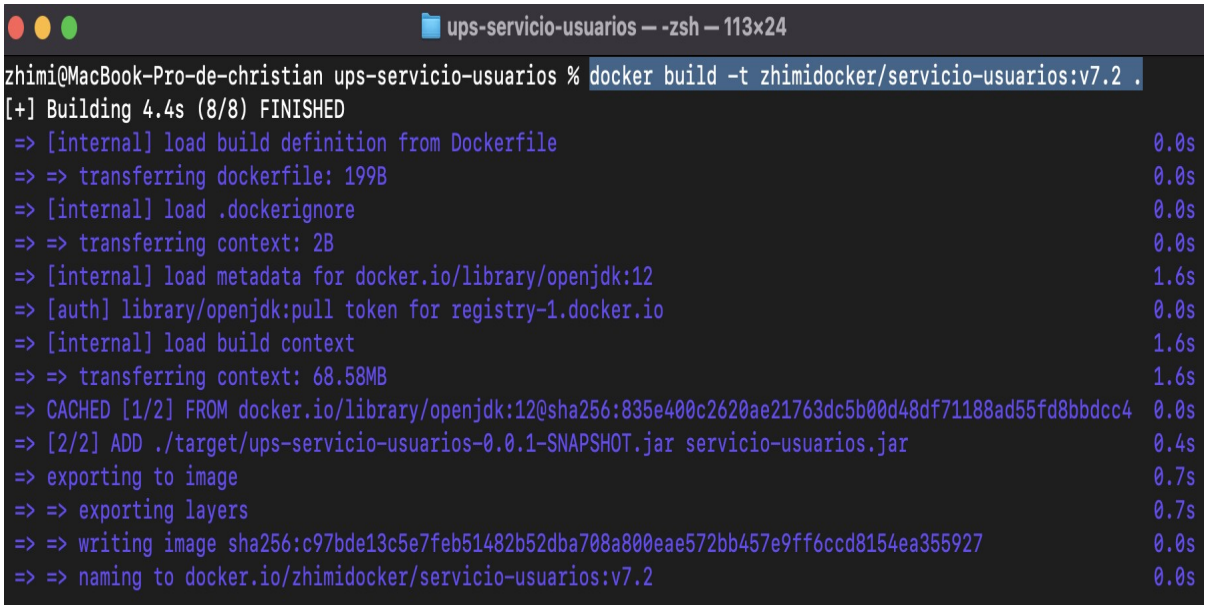



Figura 6.36: Código para generar una imagen docker.

### Docker Hub

Utilizamos el siguiente código que se muestran en la figura para subir la imagen docker generada a través del archivo dockerfile a nuestro repositorio de Docker Hub. Tener en cuenta el nombre del repositorio y de la imagen generada con su respectiva versión para proceder a subir las imágenes.

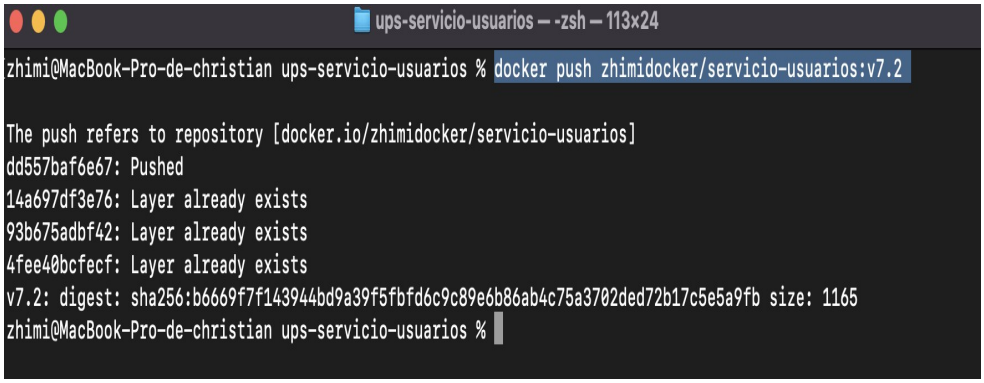



Figura 6.37: Código para subir la imagen al docker Hub.

**Docker Container**

Las imágenes docker pueden ser descargadas en cualquier plataforma que tenga instalado docker para poder ejecutar en contenedores. Para descargar la imagen del docker hub se utiliza el siguiente comando:

**( docker pull zhimidocker/servicio-usuarios:v7.2 )**

También tenemos que crear una red dentro de docker donde se van a ejecutar todos los servicios, ya que todos los servicios deben ejecutarse en la misma red para su correcto funcionamiento. El siguiente comando crea una red docker:

**( docker network create spring-microservicios )**

Luego se procede a ejecutar el contenedor con la imagen descargada, cabe recalcar que si no tiene descargado la imagen no hay problema, ya que el siguiente comando descarga automáticamente la imagen y la ejecuta en un contenedor. El siguiente comando ejecuta el servicio en un contenedor docker.

**( docker run -P –name servicio-usuarios –network spring-microservicios zhimidocker/servicio-usuarios:v7.2 )**

En la siguiente imagen se puede apreciar todos los contenedores que fueron desplegados para el funcionamiento del backend, mediante el comando **docker ps**, este comando permite listar todos los contenedores, **docker start "id del contenedor"** permite empezar un contenedor ya creado, **docker stop "id del contenedor"**, permite parar el contenedor, **docker restart "id del contenedor"** permite resetear el contenedor.

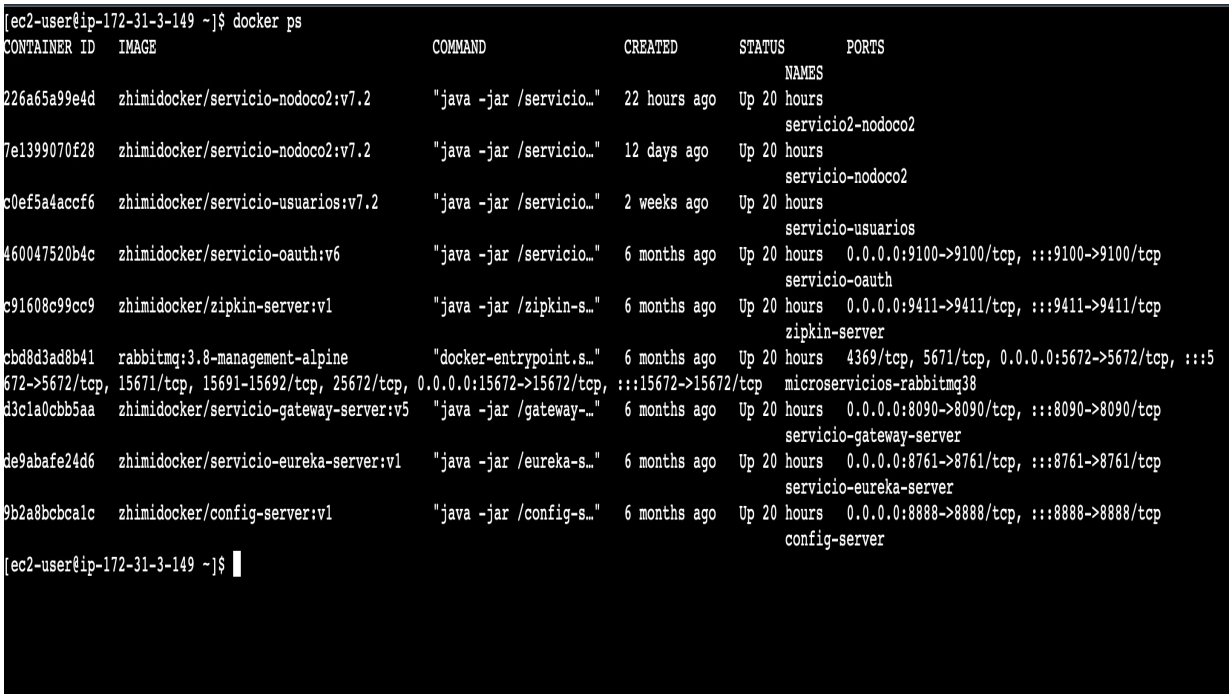



Figura 6.38: Contenedores desplegados.

### 6.3.7 FrontEnd

Frontend es la parte visible que va a interactuar con el usuario final para poder visualizar de mejor manera la información procesada de cada servicio dentro del backend, fue desarrollado con la herramienta angular con esta herramienta hemos creado la puerta de ingreso al sistema (login) donde el usuario ingresa sus credenciales para autenticarse, ya sea como un usuario administrador o un usuario normal, dependiendo cual sea el caso el sistema nos redirigirá hacia la pantalla que correspondiente.

Para la creación de un proyecto en angular utilizamos el siguiente comando

**( ng new proyectoFrontend )**

Para realizar la ejecución del proyecto creado, ingresamos a la carpeta creada y ejecutamos el siguiente comando para el despliegue de la aplicación.

**( npm run dev )**

### Comunicación del FrontEnd con el Backend

Para que nuestro frontend se comunique con nuestros microservicios tiene que hacerlo por el api gateway, por lo tanto, dentro del archivo **environment.ts** se agrega la dirección ip más el puerto de nuestro gateway, como se muestra en la siguiente imagen.

**apiURL:** Aquí ponemos la dirección ip de donde se encuentran desplegados los microservicios, en nuestro caso la siguiente dirección es la ip de mi máquina virtual en amazon más el puerto de nuestro servicio gateway.

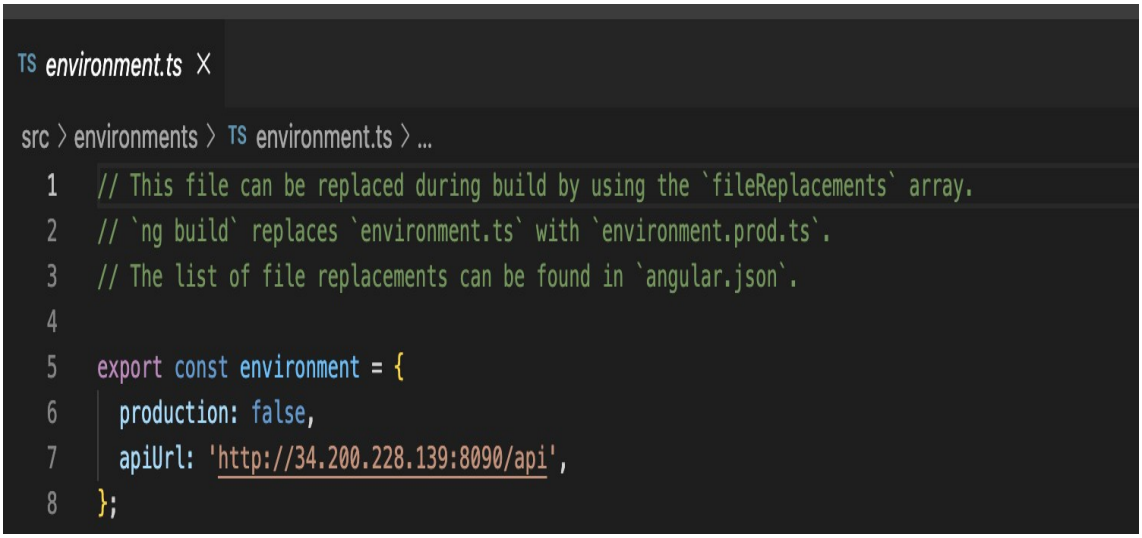



Figura 6.39: Comunicación del frontend con la Api Gateway.

### Comunicación del FrontEnd con el servicio usuario

El archivo **user.service.ts** nos permite establecer la conectividad de nuestro microservicio usuarios con el frontend para poder filtrar y visualizar la información recuperada del backend, en

este archivo se declara el endpoint del servicio usuario, y se implementa los métodos (get, post, delete, put) necesarios para poder abstraer la información de acuerdo con los métodos desarrollados en el backend. En la siguiente imagen se observa el archivo **user.service.ts** desarrollado en angular.

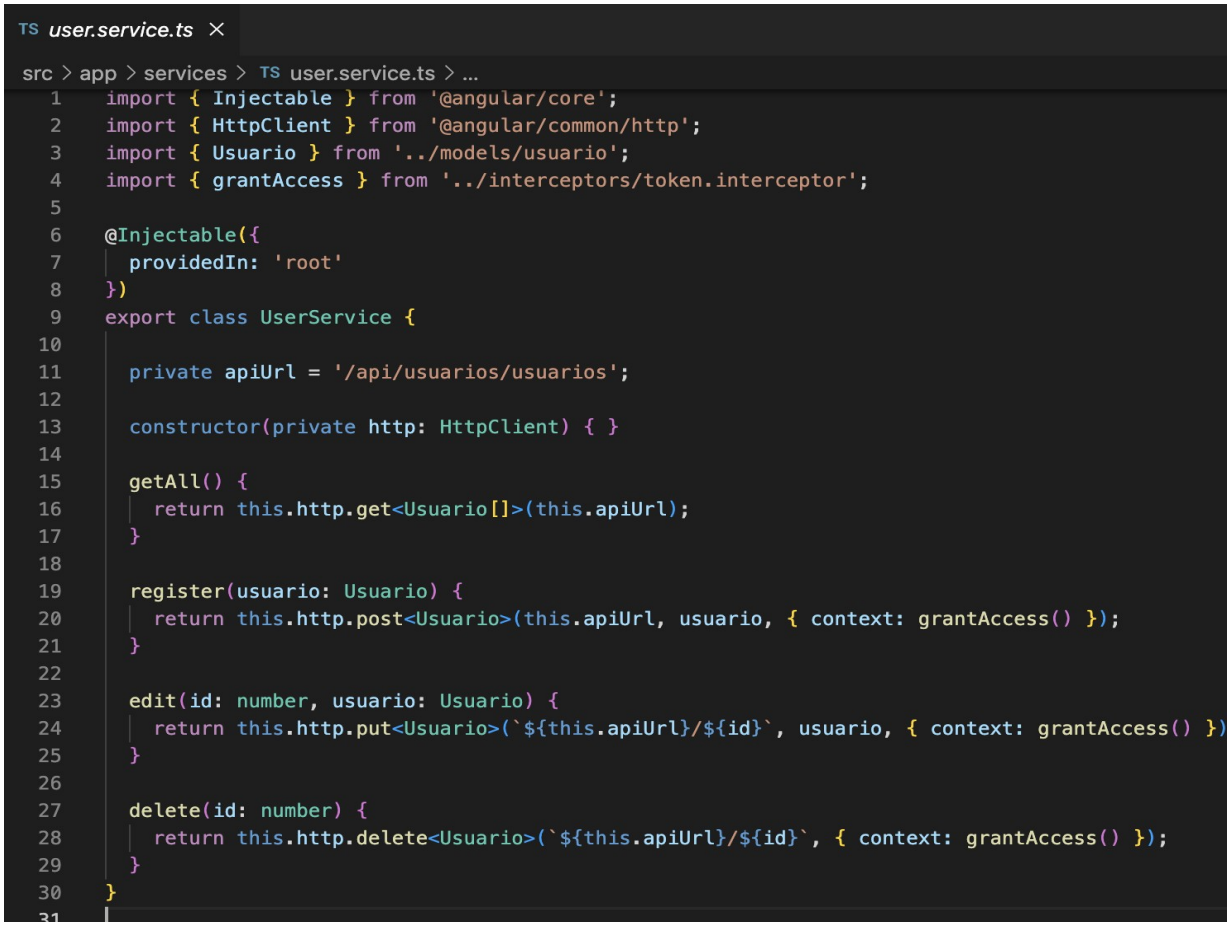



Figura 6.40: Comunicación con el servicio usuarios.

### Comunicación del FrontEnd con el servicio OAUTH

Se creo un nuevo archivo dentro de angular **auth.service.ts** para el servicio oauth, primero se agrega el endpoint del microservicio oauth donde se maneja toda la seguridad del sistema mediante tokens, nuestro backend está protegido por un usuario y rol, tenemos que autorizar y validar el usuario que ingrese al sistema. Para esto el usuario solicita un token de acceso de acuerdo con sus credenciales y es un usuario registrado, procedemos a almacenar el token, para

que realice las peticiones necesarias enviando el token almacenado.

En la siguiente figura se observa el código que fue implementado para realizar una solicitud **POST** a nuestro backend enviando el headers del token, y poder verificar el acceso a los recursos dependiendo el usuario ingresado, en la imagen 6.16 se observa la arquitectura que permite generar y tener acceso a los recursos mediante el token generado.


Figura 6.41: Comunicación con el servicio oauth.

### Comunicación del FrontEnd con el servicio NodoCo2

El archivo **nodoco2.service.ts** nos permite establecer la conectividad con nuestro microservicio NodoCo2 con el frontend para poder abstraer, filtrar y visualizar la data en gráficas mediante su interfaz, en este archivo se declara el endpoint del servicio y se implementa los métodos **GET** de acuerdo con los métodos desarrollados en el backend, el cual internamente se encarga

de procesar y realizar la petición recibida, en este caso el backend solicitara a la “base de datos MYSQL” la data necesaria de acuerdo con los filtros enviados como parámetros. La conexión con la base de datos está establecida en el archivo "servicio-nodoco2-prod.properties" de gitlab tal como se observa en la imagen 6.26.

En la siguiente imagen se observa el archivo **nodoco2.service.ts** desarrollado en angular.

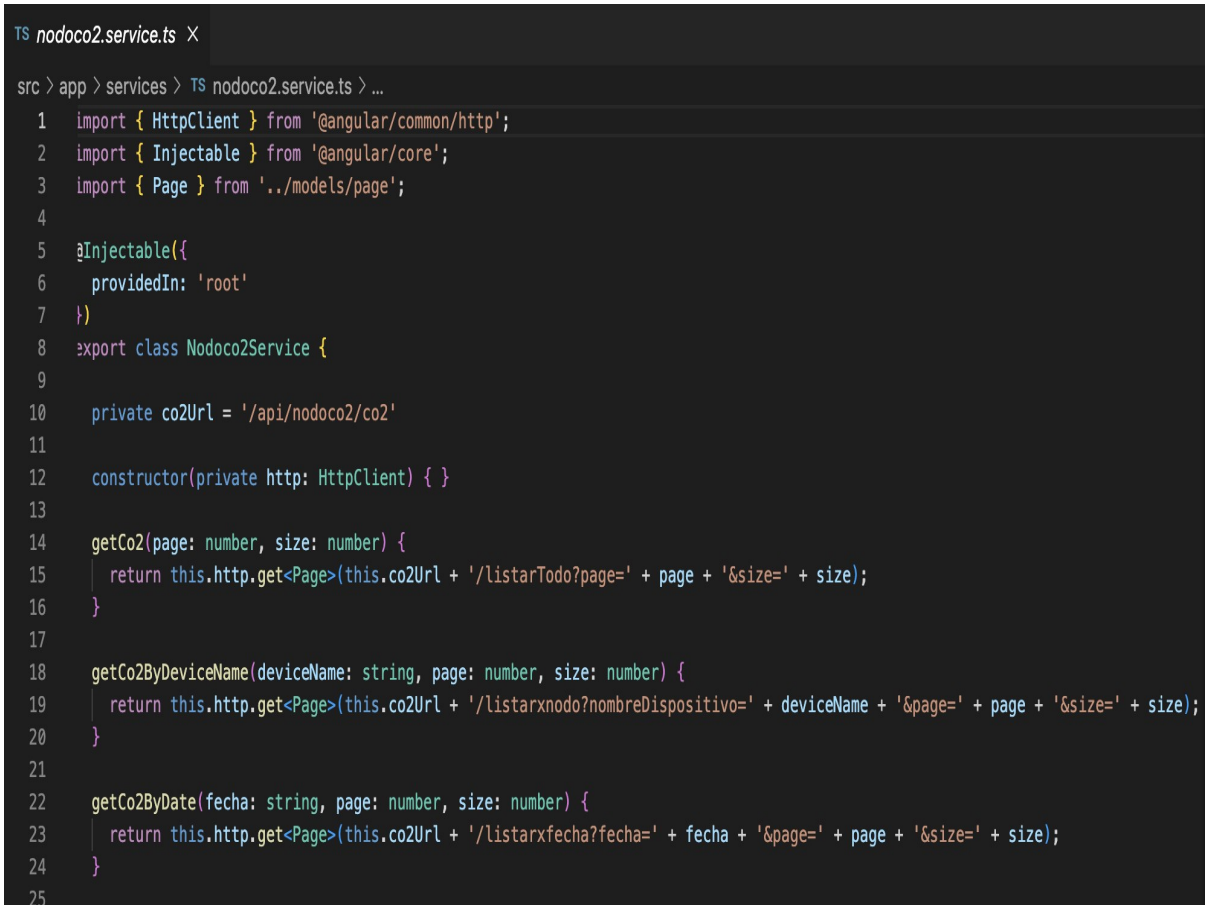



Figura 6.42: Comunicación con el servicio NodoCo2.

# Capítulo VII

# Resultados

En este capítulo se presentan todos los resultados obtenidos en el desarrollo de la plataforma IoT basado en una arquitectura de Microservicios.

Las pruebas de conexión y funcionamiento entre microservicios nos brindaron resultados favorecidos, gracias a la utilización de herramientas tecnológicas muy funcionales para arquitecturas de microservicios.

## 7.1 Hardware y Software

El Hardware utilizado para el desarrollo de este proyecto es:

- Macbook Pro 2018
- Sistema MacOs Monterey Versión 12.6
- Memoria Ram: 8GB
- Procesador: 2,3 GHz Intel Core i5 de 4 núcleos

Las herramientas de Software utilizadas para las pruebas de conexión y funcionamiento son:

- **Eureka Server:** Mediante su interfaz nos permite visualizar los servicios registrados correctamente y comprobar la escalabilidad.
- **Zipkin Server:** Gracias a su interfaz se puede revisar las trazas de cada servicio.
- **RabbitMQ:** Mediante su interfaz se puede revisar los servicios que se registren como productores y consumidor.
- **Postman:** Nos permite realizar pruebas de funcionalidad de los métodos implementados en cada servicio.

## 7.2 Resultados de Eureka Server

Eureka server es un complemento de Spring Boot, el servidor eureka conoce todas las aplicaciones cliente que se registran al servidor con su puerto y dirección IP. El servidor eureka nos permite identificar y conectar cada microservicio, porque todas las instancias de servicio en el registro tienen que enviar latidos del corazón para mantener sus registros actualizados.

Primero se despliega el servidor de configuración, ya que contiene las configuraciones para cada servicio. Luego procedemos a desplegar el servidor eureka, para que pueda registrar a los clientes que se conecten al servidor y por último desplegamos todos los microservicios clientes que vamos a necesitar para este proyecto.

Como resultado obtuvimos que todos los servicios se desplegaron y se registraron correctamente a eureka. Se puede apreciar este resultado en la siguiente figura:

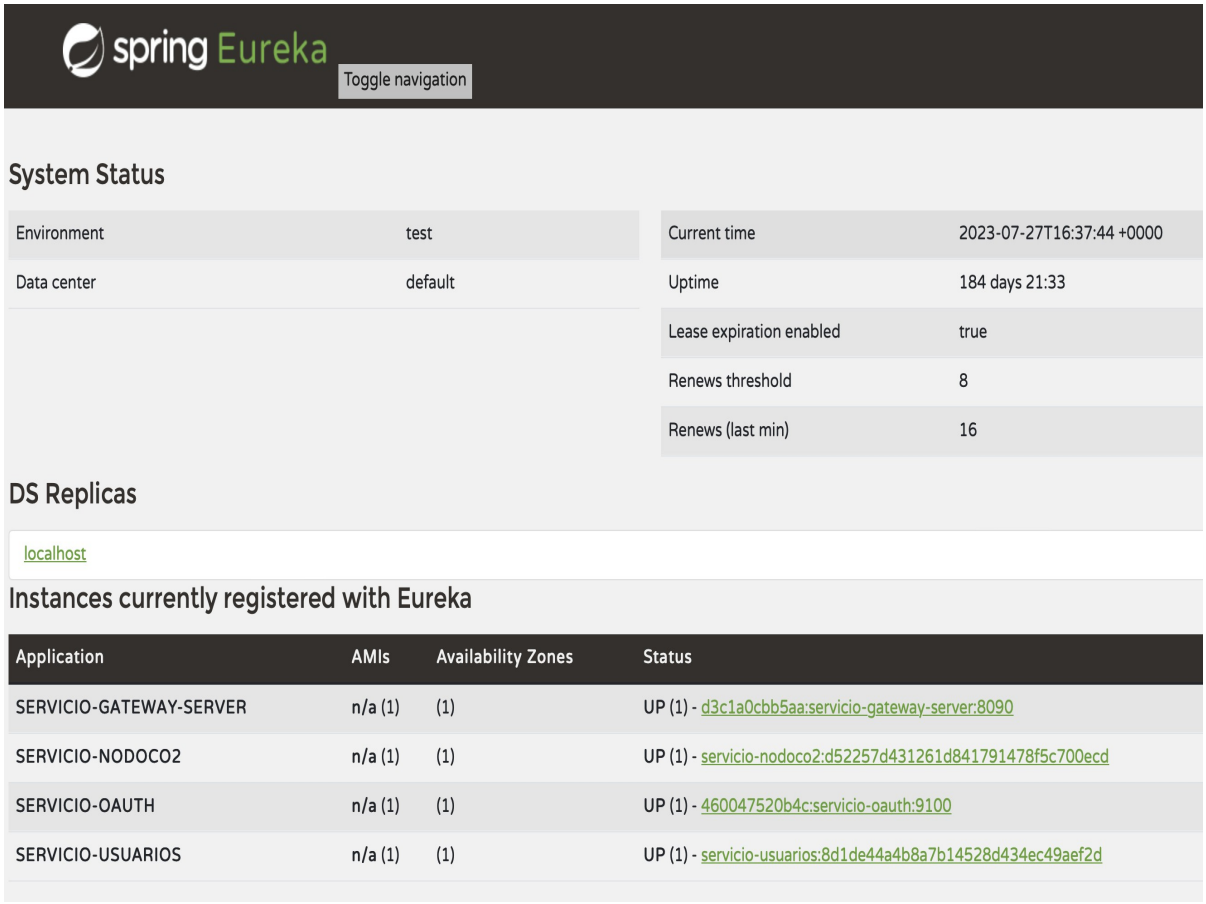



Figura 7.1: Microservicios Registrados en Eureka Server.

Se puede visualizar en la interfaz de eureka los nombres de todos los microservicios con su respectivo puerto, que se registraron en el servidor. Los servicios que no muestran su puerto son porque fueron configurados con puertos dinámicos, esto permite ocultar el puerto por seguridad y para poder levantar más instancias del mismo servicio.

### 7.2.1 Pruebas de Escalabilidad en Eureka Server

El servidor de nombres "Eureka server" nos brinda características funcionales para el desarrollo de microservicios, una de ellas es la alta disponibilidad y la escalabilidad que nos permite desplegar más instancias de servicios para poder obtener una mejor respuesta a las peticiones que se les realiza, haciendo que el sistema sea robusto.

Para levantar más instancias tenemos que volver a ejecutar el mismo comando docker run, siempre

y cuando tenga un puerto dinámico, ya que con cada ejecución se registrará con un puerto diferente para el uso del microservicio.

En este caso como estamos utilizando docker, se tiene que ejecutar la siguiente línea de código:

**( docker run -P –name servicio-usuarios –network spring-microservicios zhimidocker/servicio-usuarios:v7.2 )**

Donde **-P** hace referencia que se ejecutara en un puerto aleatorio, **–name** es el nombre del contenedor que va a ejecutar el servicio, por lo tanto, no puede repetirse el nombre para poder levantar una nueva instancia del microservicio y **–network** quiere decir a que red docker va a pertenecer el contenedor, entonces debe de ser la misma red que contiene todos los servicios para que se pueda ejecutar como una segunda instancia. En la siguiente imagen se puede visualizar el resultado de levantar varias instancias de un microservicio:

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| SERVICIO-GATEWAY-SERVER | n/a (1) | (1) | UP (1) - 172.16.208.143:servicio-gateway-server:8090 |
| SERVICIO-NODOCO2 | n/a (2) | (2) | UP (2) - servicio-nodoco2:567ce0a92a0b0d4caee9a09e642f758c , servicio-nodoco2:851d7ace6d91ad94fc508751e14ba8c2 |
| SERVICIO-OAUTH | n/a (1) | (1) | UP (1) - 172.16.208.143:servicio-oauth:9100 |
| SERVICIO-USUARIOS | n/a (3) | (3) | UP (3) - servicio-usuarios:780a5a8ec908f433d439cb6604b843cb , servicio-usuarios:a0dfa34df6fae6ad5516e37e62a90c6a , servicio-usuarios:98013cf8fe10f39f6a89e636678558f4 |

Figura 7.2: Escalabilidad de microservicios.

En la imagen anterior 7.2 nos permite observar cómo fueron levantadas 2 instancias del microservicio nodoCo2 y 3 instancias del microservicio usuarios.

En la siguiente imagen 7.3 visualizamos el resultado de bajar "down" una instancia, para bajar la instancia dentro de docker se ejecutó la siguiente línea de código:

**docker stop "id del contenedor"**

Si deseamos volver a levantar la instancia que fue dada de baja solo tenemos que ejecutar.

**docker start "id del contenedor bajado"**

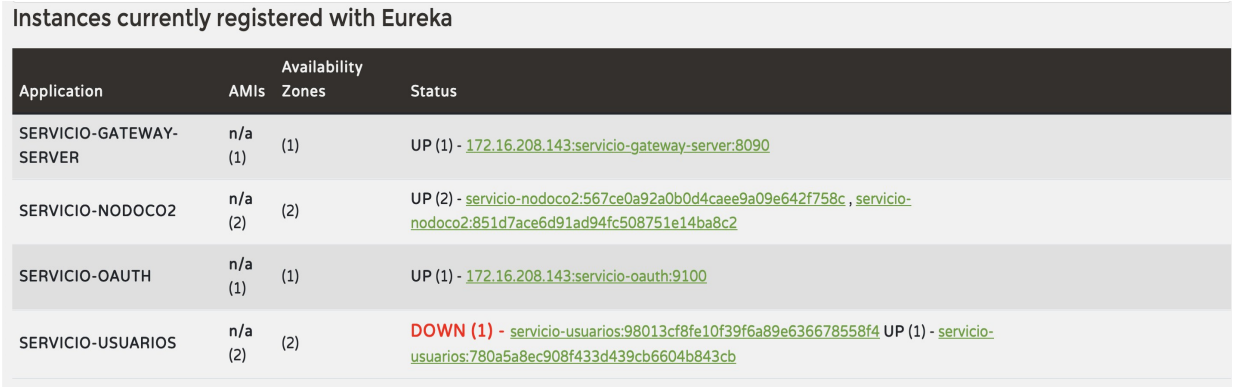



Figura 7.3: Bajando un servicio escalado en Eureka.

## 7.3 Resultados con RabbitMQ

Mediante la interfaz que nos brinda el broker de mensajes "RabbitMQ" se observa los microservicios que se registran como productores de mensajes para que el broker pueda transmitir al consumidor zipkin estos mensajes, tal como se observa en el diagrama de la imagen 6.33.

Primero desplegamos el broker RabbitMQ y después procedemos a desplegar los microservicios para que se suscriban a nuestro broker instalado, tal como observamos en la siguiente imagen:

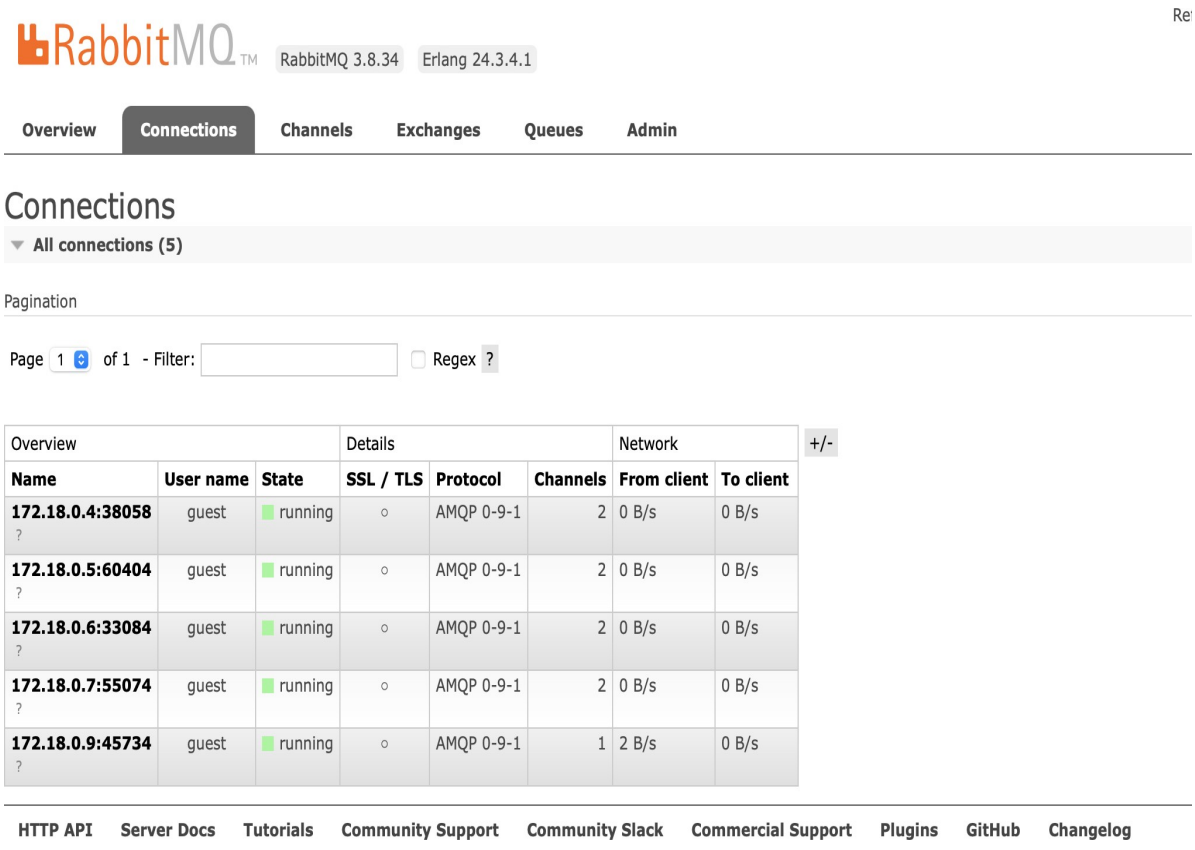



Figura 7.4: Servicios suscritos en RabbitMQ.

Luego probamos que el consumidor zipkin se suscriba a nuestro broker, desplegando el servidor zipkin después de los microservicios y comprobamos en la interfaz de rabbit que este suscrito como un consumidor dentro de un canal para que pueda consumir la cola de mensajes, este resultado se observa en la siguiente imagen:

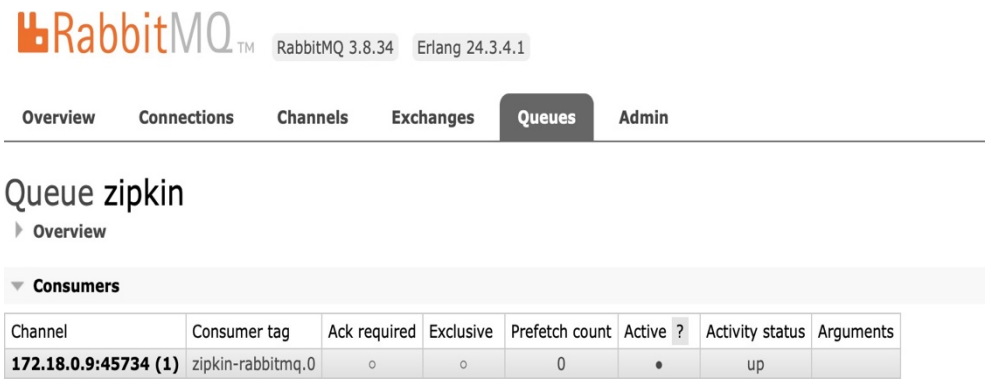



Figura 7.5: Zipkin suscrito como consumidor.

En la siguiente imagen 7.6 se observa los canales que nos proporcionan una vía de comunicación para los servicios conectados, para que puedan enviar los mensajes de un gestor de colas a otro, cuando se cierra una conexión, también se cierran los canales que tiene asociados.

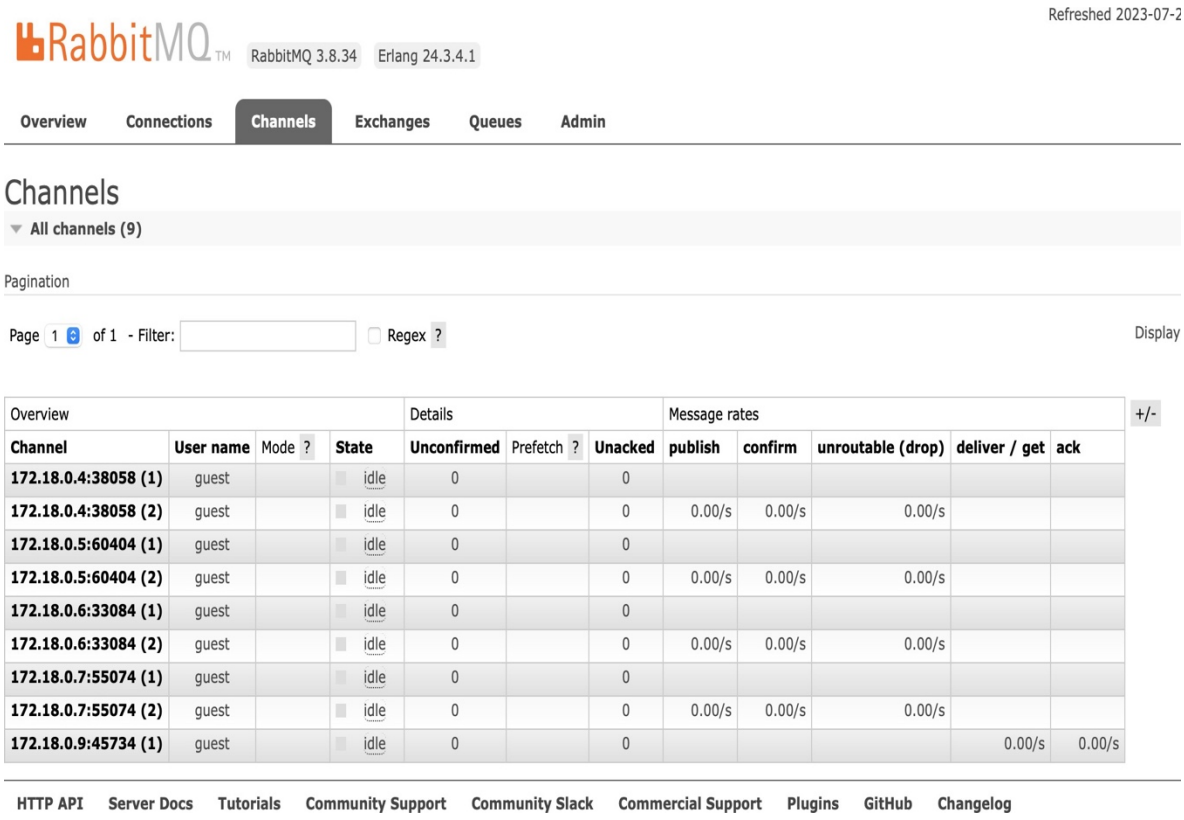



Figura 7.6: Canales creados para la comunicación.

## 7.4 Pruebas realizas con el cliente Postman

Gracias al cliente postman, se realiza las pruebas de conexión y funcionamiento de nuestro backend, para poder verificar si tenemos algún problema con la comunicación de los servicios, ya que nos aporta una interfaz donde se realiza solicitudes post, get, put y delete, y muchas más.

### 7.4.1 Pruebas de comunicación de chirpstack con el backend.

Una vez que desplegamos nuestro microservicio nodoCo2, el cual es el encargado de recibir y procesar la data que envía el servidor chirpstack verificamos que se ejecute sin problemas, en la siguiente imagen 7.7 se puede apreciar que el servicio de integración con chirpstack se ejecutó correctamente, entonces nuestro backend está listo para recibir y almacenar la data de los nodos.

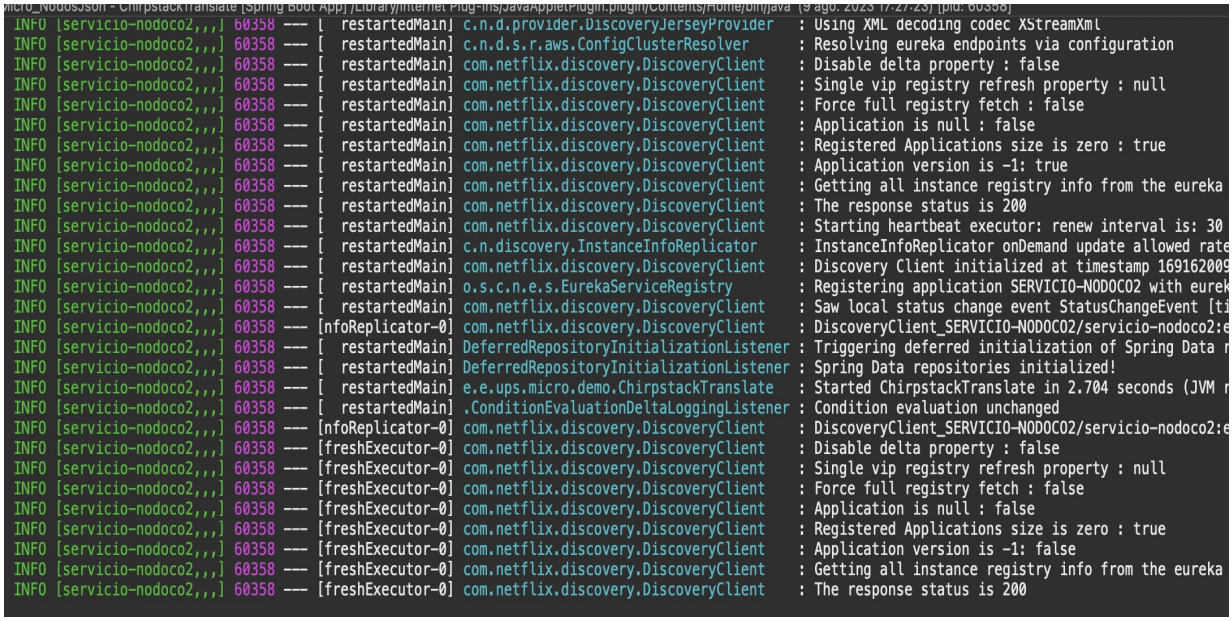



Figura 7.7: Microservicio NodoCo2 desplegado exitosamente.

### 7.4.2 Pruebas de envió de la data mediante el cliente postman

Realizamos unas pruebas personalizadas desde el cliente postman, enviamos la data en formato json para comprobar que nuestro microservicio este recibiendo y guardando la data correctamente en su respectiva base de datos.

La siguiente imagen muestra cómo se envía la data en formato json, desde el cliente postman, el json que se muestra en la imagen, es un ejemplar del json original que envía el servidor chirpstack, con todos sus atributos que genera el nodo conectado al chirpstack, tal como el id y nombre de la aplicación, el nombre del dispositivo, el número de placa, la fecha, un object JSON y más atributos, pero nosotros solo vamos a almacenar los atributos necesarios como el nombre del dispositivo, la fecha, y el object JSON que contiene los valores de co2A Y del co2B.

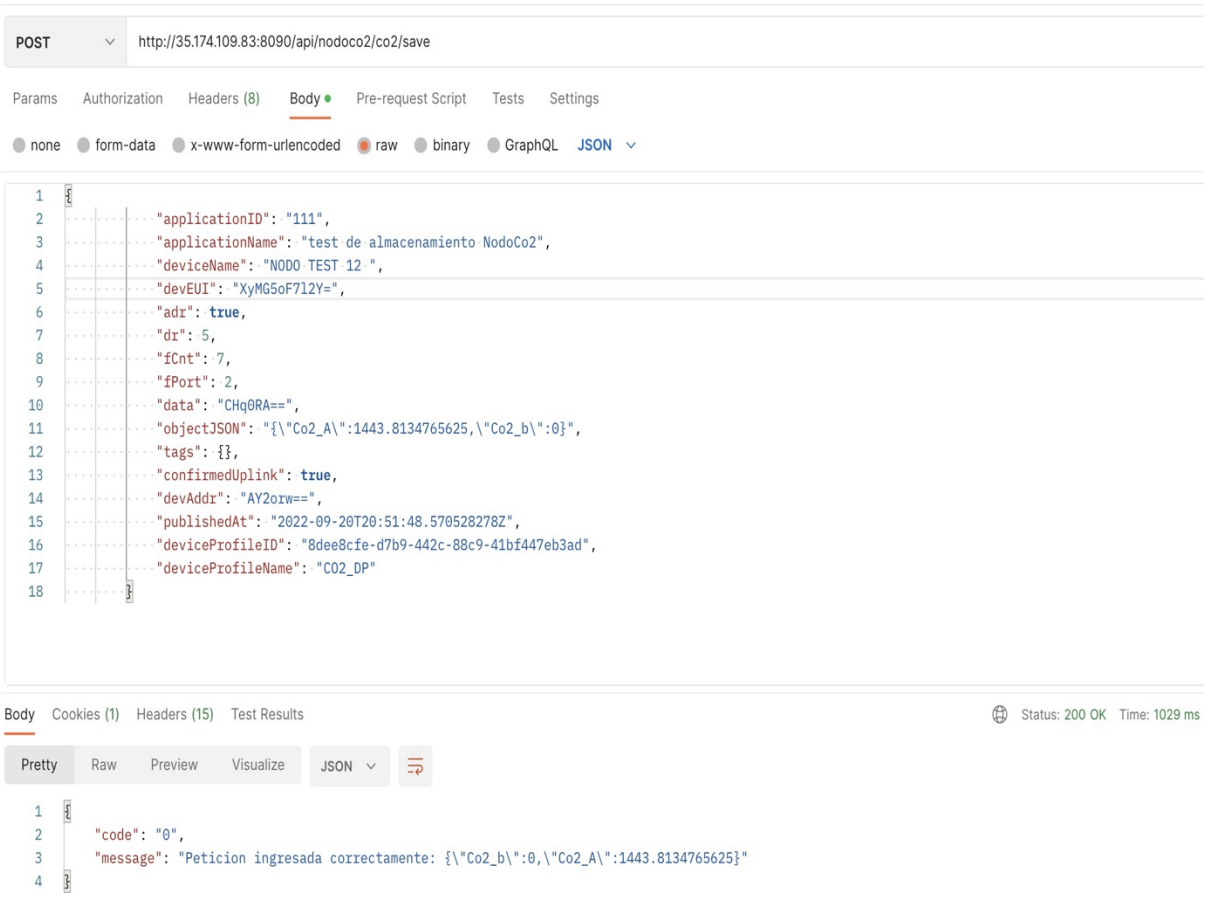



Figura 7.8: Data enviada desde Postman.

Ingresamos a nuestra base de datos MYSQL, realizamos una búsqueda por el nombre del dispositivo "device name=NODO TEST" y verificamos que la data se guardó exitosamente solo con las variables necesarias para nuestro procesamiento gráfico, tal como está en la siguiente imagen:

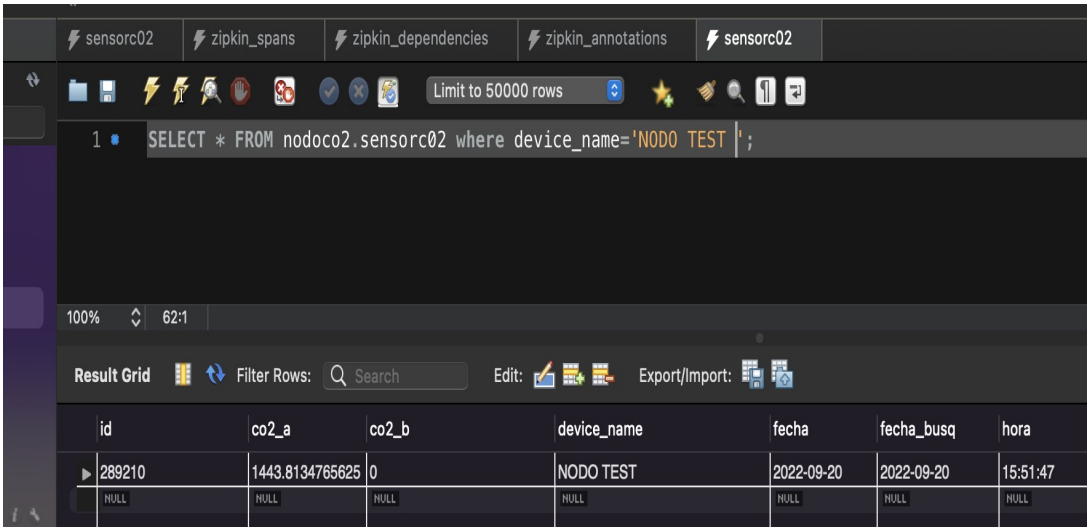
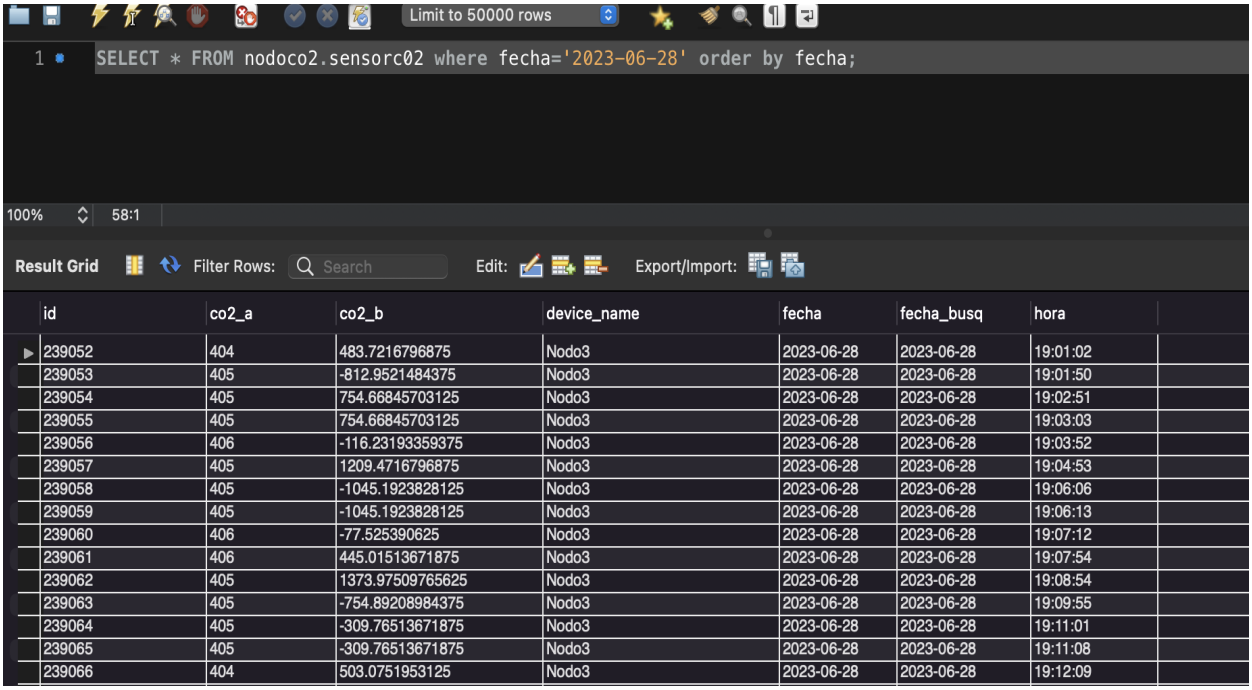



Figura 7.9: Data almacenada exitosamente.

### 7.4.3 Resultados del envió de la data desde chirpstack

Nuestro microservicio está recibiendo la data correctamente y enviándole a la base de datos sin problemas. Después de realizar las pruebas, se puede saber que la data que nos está llegando y guardándose es un archivo json de chirpstack, para verificar la data realizamos una búsqueda dentro de nuestra base de datos con la fecha actual para visualizar la data enviada desde el servidor chirpstack.

La siguiente imagen nos muestra la data almacenada exitosamente en nuestra base de datos MySQL.




| id | co2_a | co2_b | device_name | fecha | fecha_busq | hora |
|---|---|---|---|---|---|---|
| 239052 | 404 | 483.7216796875 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:01:02 |
| 239053 | 405 | -812.9521484375 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:01:50 |
| 239054 | 405 | 754.66845703125 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:02:51 |
| 239055 | 405 | 754.66845703125 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:03:03 |
| 239056 | 406 | -116.23193359375 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:03:52 |
| 239057 | 405 | 1209.4716796875 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:04:53 |
| 239058 | 405 | -1045.1923828125 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:06:06 |
| 239059 | 405 | -1045.1923828125 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:06:13 |
| 239060 | 406 | -77.525390625 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:07:12 |
| 239061 | 406 | 445.01513671875 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:07:54 |
| 239062 | 405 | 1373.97509765625 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:08:54 |
| 239063 | 405 | -754.89208984375 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:09:55 |
| 239064 | 405 | -309.76513671875 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:11:01 |
| 239065 | 405 | -309.76513671875 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:11:08 |
| 239066 | 404 | 503.0751953125 | Nodo3 | 2023-06-28 | 2023-06-28 | 19:12:09 |

Figura 7.10: Data almacenada de la integración de chirpstack con el backend.

### 7.4.4 Pruebas de Autorización para los recursos

Para comprobar las funcionalidades del servicio usuarios se realizó las pruebas con el cliente postman, realizamos las peticiones directo a nuestra API Gateway de acuerdo con las funcionalidades implementadas en los microservicios.

Dentro de nuestro servidor de autorización establecimos la configuración para dar acceso a los recursos de los microservicios, de acuerdo del tipo de rol que tenga el usuario autentificado.
A continuación, describimos los permisos que establecimos en el backend:

- **Acceso Público:** Sin autenticación solo se puede listar todos los usuarios existentes.
- **Usuario Normal:** Permitimos el acceso para listar todos los usuarios existentes y para poder buscar por el número de id del usuario.
- **Usuario Administrador:** Permitimos el acceso a todos los recursos en general, solo el administrador tiene permisos para poder crear y eliminar un usuario.

**Pruebas con el Acceso Público**

- Mediante el cliente postman vamos a enviar una petición GET a nuestro api gateway del backend, la cual comunica con el recurso del microservicio que estamos solicitando, y nos da como respuesta la lista de usuarios existentes, tal como se puede apreciar en la imagen 7.11.

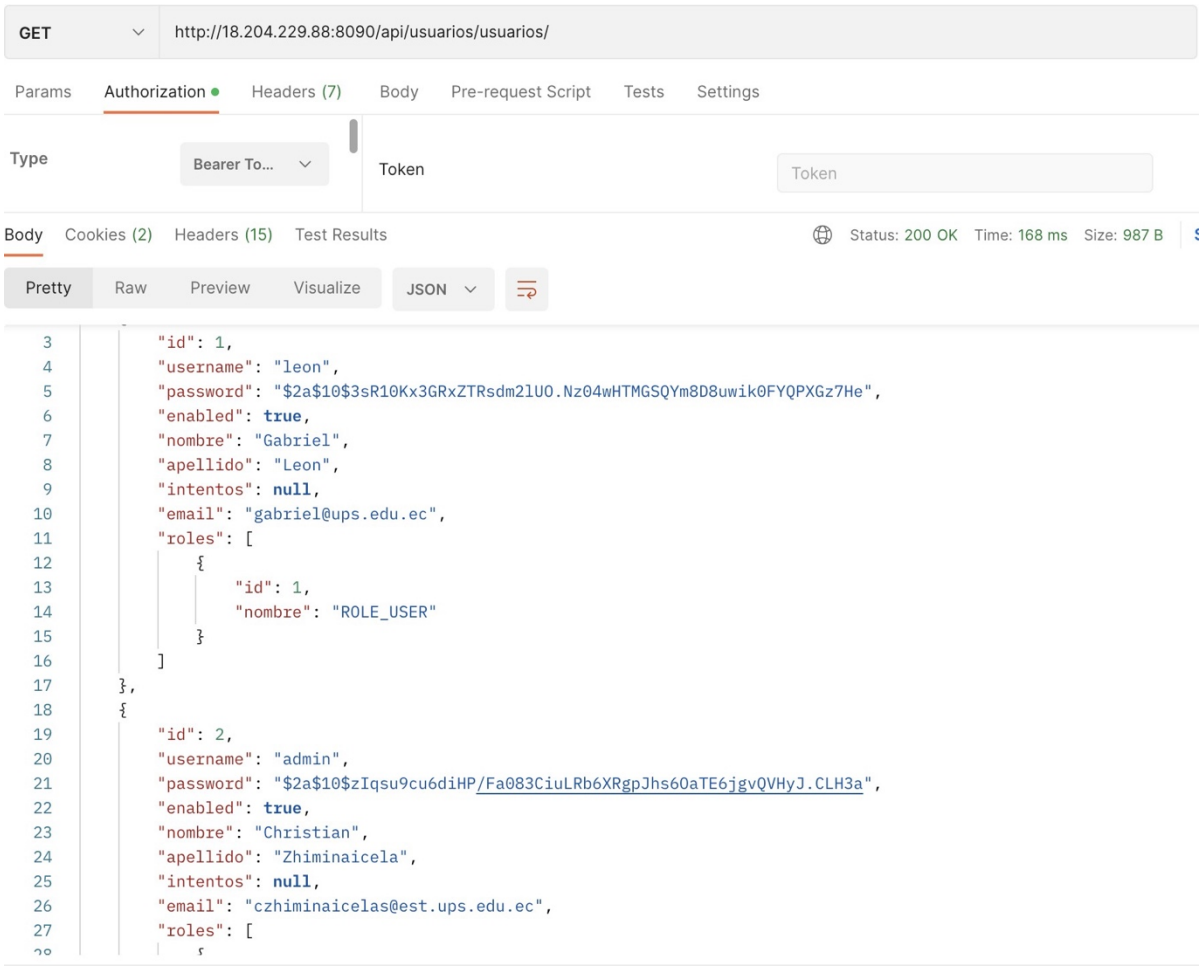



Figura 7.11: Listado de usuarios existentes.

- Verificamos que el recurso está protegido, intentando realizar una petición GET para listar al usuario por su id, y vemos que tenemos como respuesta un estado 401 no autorizado, tal como nos muestra la siguiente imagen 7.12.

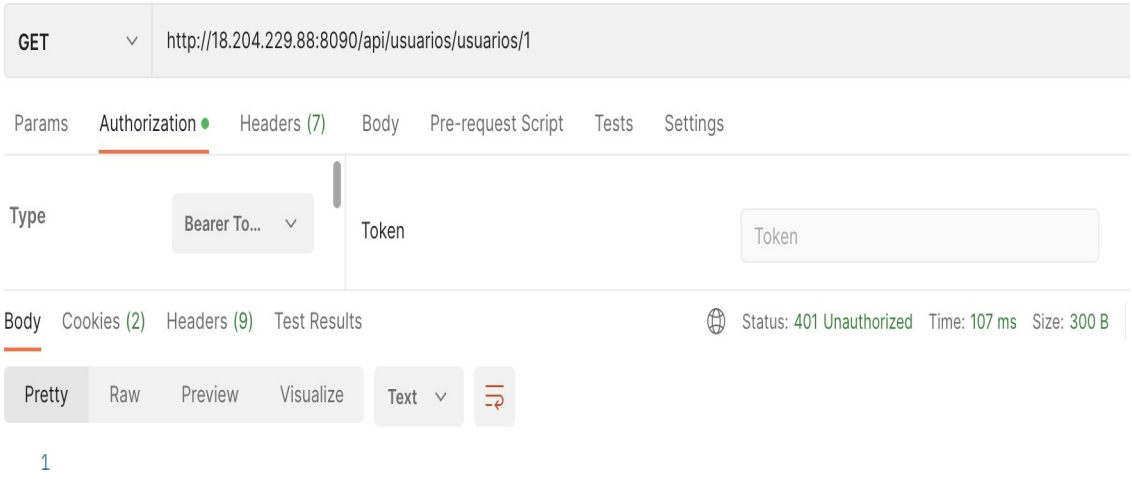



Figura 7.12: Listado por id no autorizado.

**Pruebas con el Usuario Normal**

- 1. Vamos a realizar la autenticación de usuario mediante nombre de usuario, la clave y el tipo de seguridad "password". Esto nos permite generar un token para poder acceder a los recursos establecidos para este usuario, para este proceso vamos a enviar una petición POST a nuestro endpoint del backend, tal como se puede visualizar en la siguiente imagen 7.13:

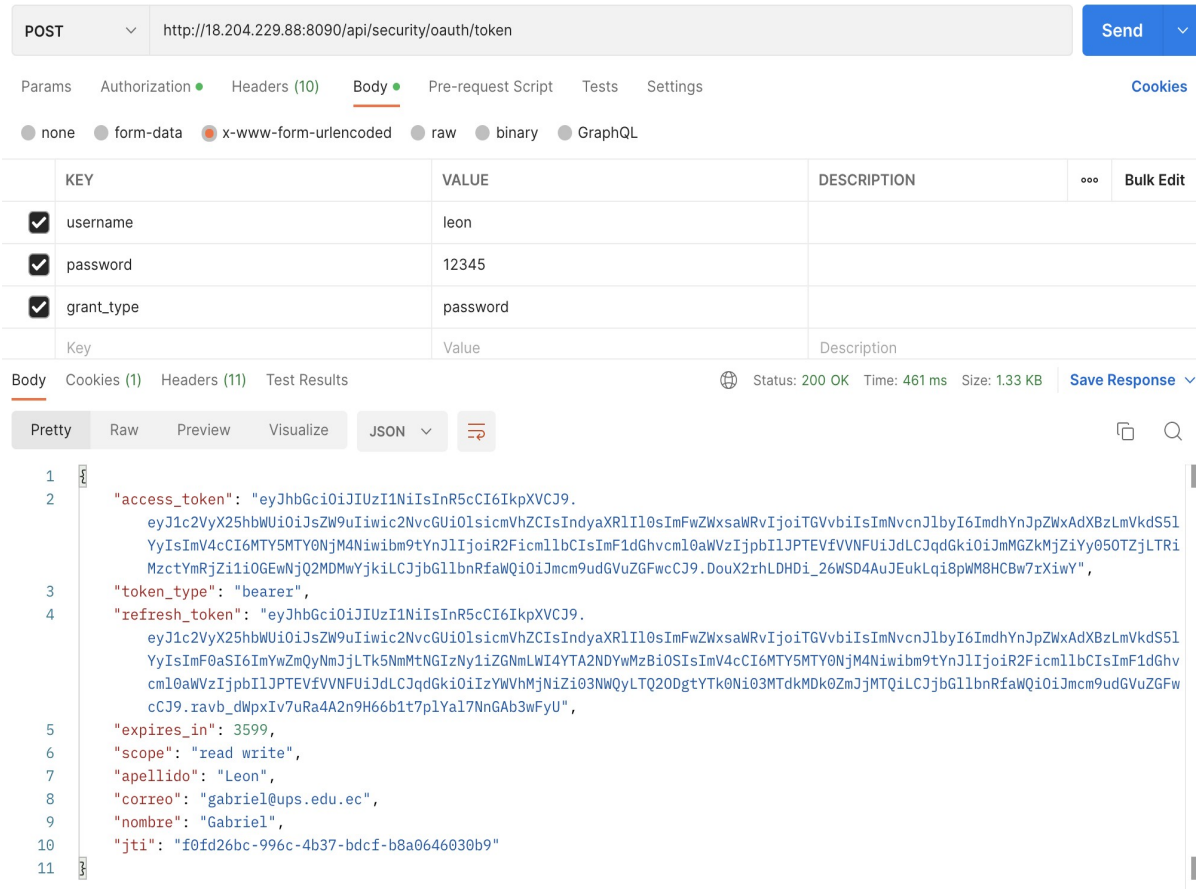



Figura 7.13: Generando token mediante la autenticación de usuario.

- 2. Validamos el token dentro de la opción autorización, escogemos el tipo de seguridad que seria "Bearer Token" e ingresamos el token generado, para autorizar los recursos de acuerdo con el usuario autentificado, en este caso nos autoriza poder listar un usuario por el id, el resultado se puede visualizar en la siguiente imagen 7.14:

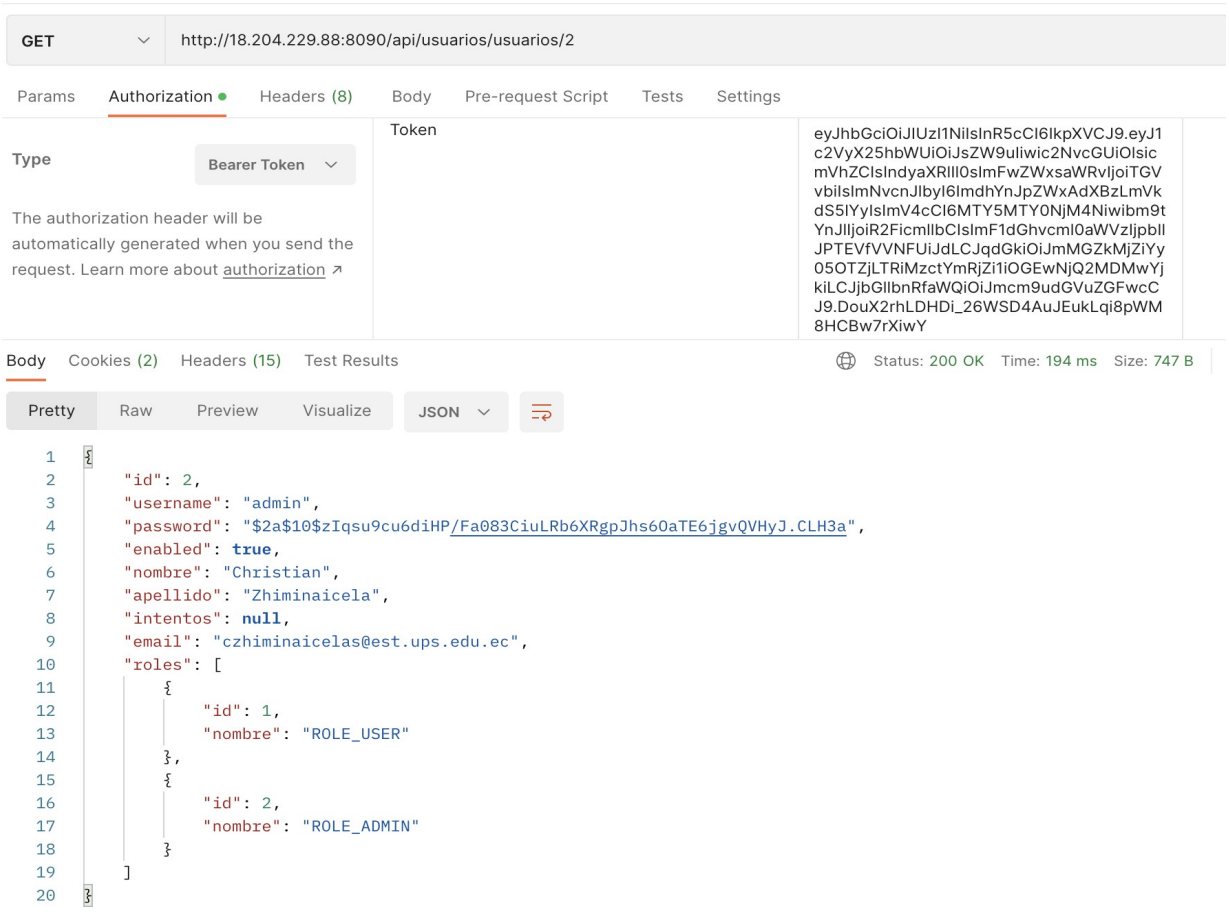



Figura 7.14: Token Autorizando para listar por usuarios por id.

- 3. Realizamos prueba de protección de recursos para diferentes usuarios, como ya ingresamos el token generado, para autorizar los recursos de acuerdo al usuario autentificado, en este caso no permite autorizar las operaciones de crear y eliminar usuarios, ya que es un recurso que está permitido solo para administradores, enviamos una petición POST con su respectivo cuerpo a nuestro endpoint de usuarios y vemos que tenemos como respuesta un estado 401 no autorizado, tal como se puede visualizar en la siguiente imagen 7.15:

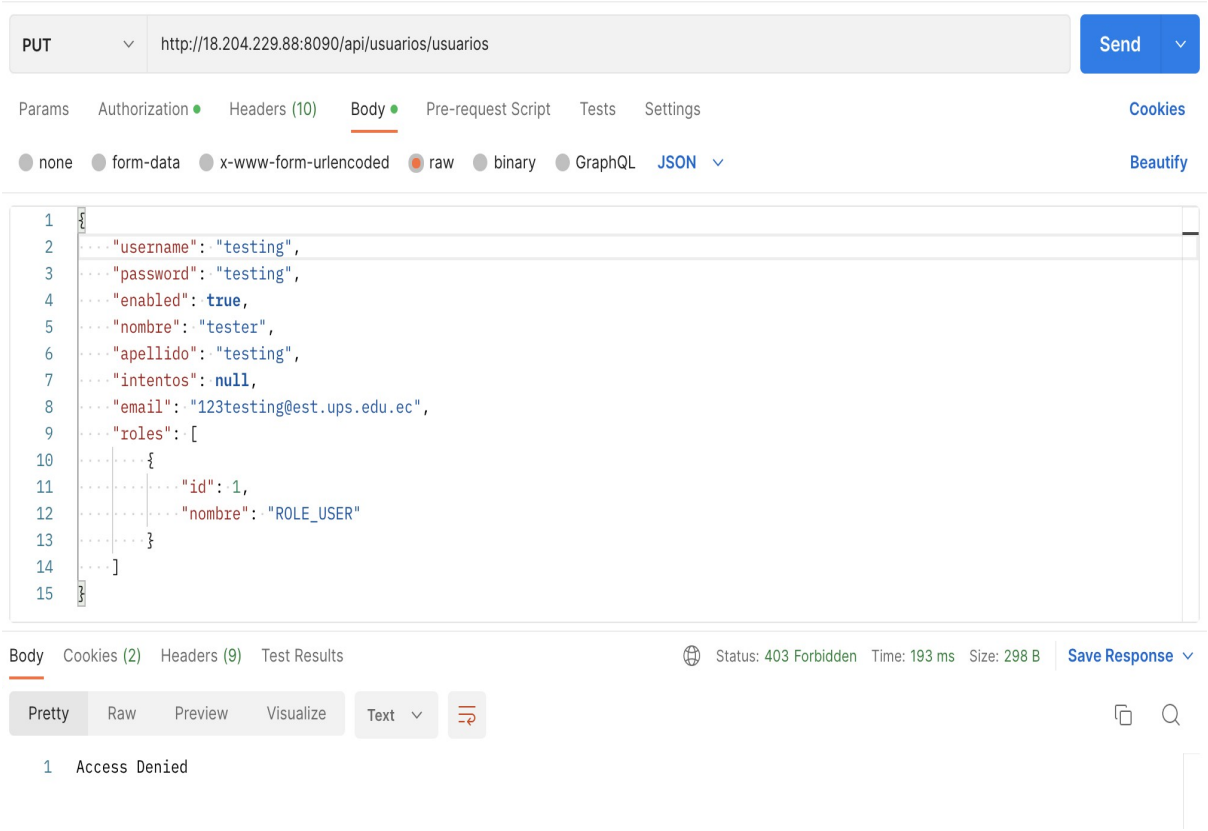



Figura 7.15: Autorización denegada para la creación de usuario.

**Pruebas con el Usuario Administrador**

- 1. Realizamos la autenticación del usuario administrador, mediante nombre de usuario, la clave y el tipo de seguridad "password". Esto nos permite generar un token para poder acceder a todos los recursos en general, ya que es un usuario administrativo, para este proceso vamos a enviar una petición POST a nuestro endpoint del backend, tal como se puede visualizar en la siguiente imagen 7.16:

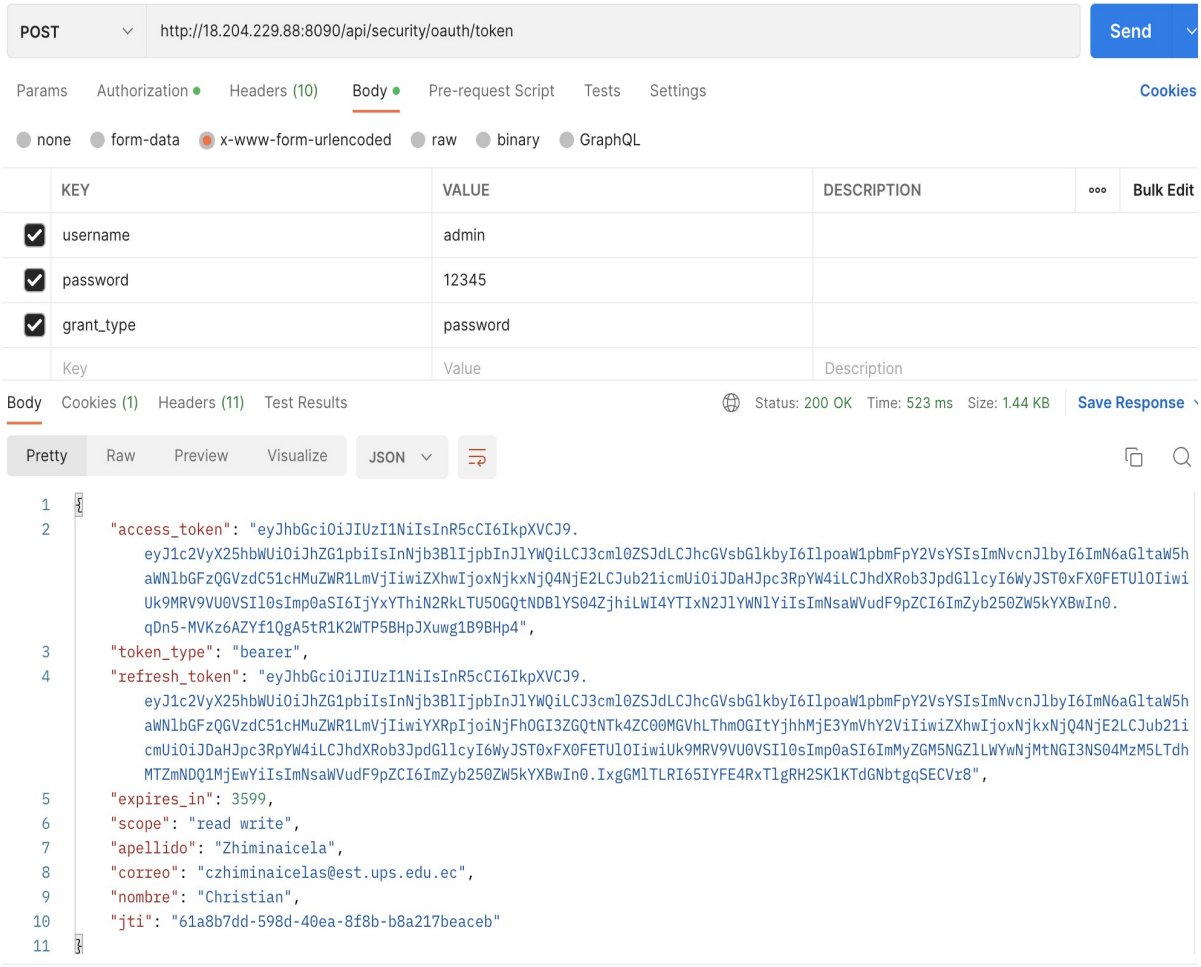



Figura 7.16: Token generado con usuario administrador.

- 2. Ingresamos el token generado, para autorizar los recursos al usuario administrador, validamos el token y procedemos a realizar las operaciones como crear, buscar, eliminar y actualizar usuarios. Procedemos a crear un nuevo usuario con sus datos y con un rol de usuario normal, mediante una petición POST al endpoint del backend. Como resultado obtenido tenemos un estado 200 ok que nos dice que el usuario fue creado exitosamente, en la siguiente imagen 7.17 se puede visualizar los detalles.

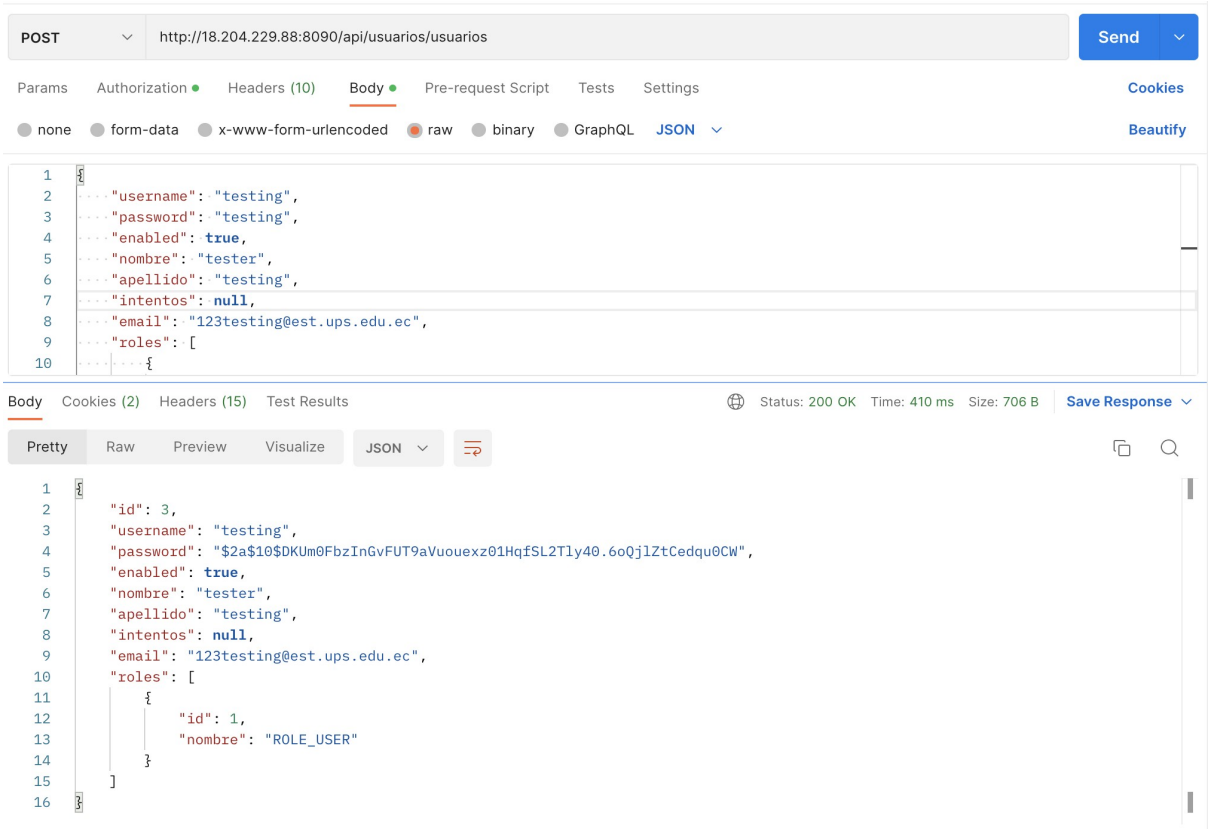



Figura 7.17: Usuario Creado Exitosamente.

- 3. Con el mismo token ingresado procedemos a realizar una petición PUT al endpoint de nuestro backend, esta petición actualiza la información del usuario ingresando el parámetro id. Actualizamos el username obteniendo un resultado exitoso "estado: 200 ok", tal como se puede visualizar en la imagen 7.18:

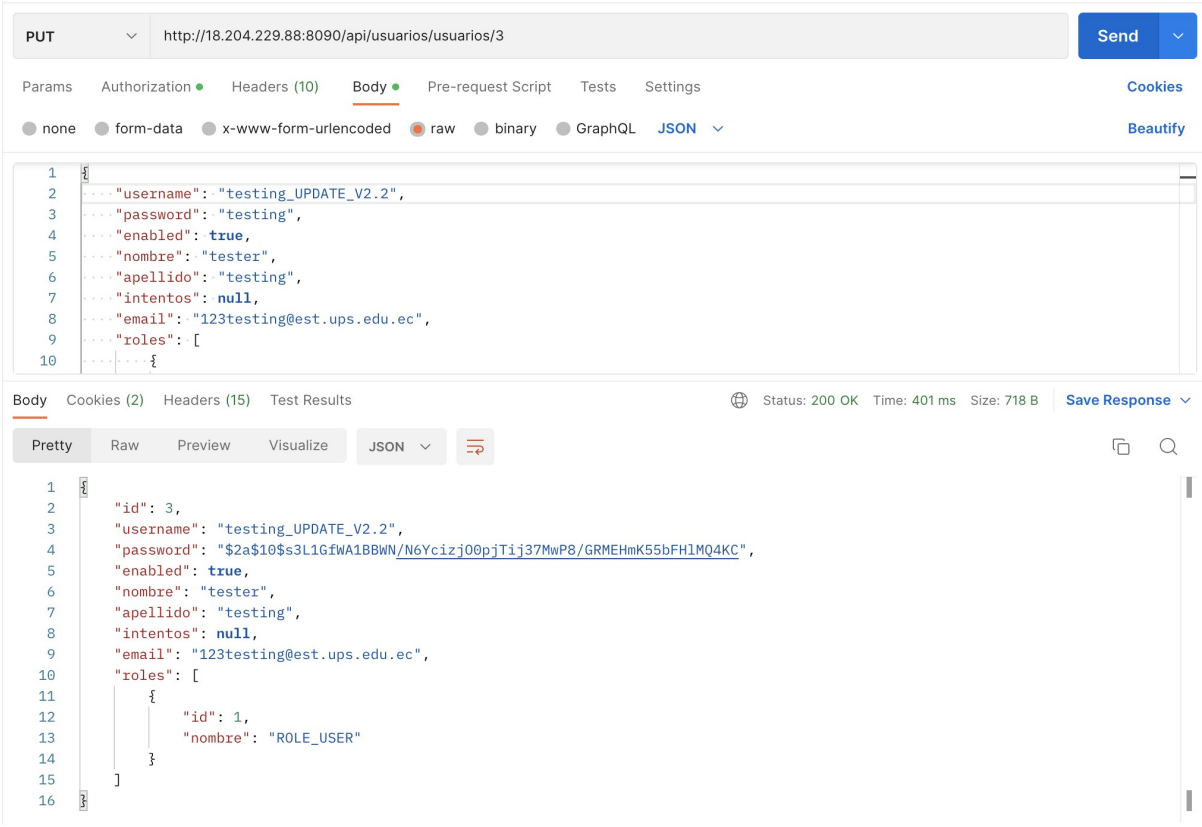



Figura 7.18: Usuario Actualizado Exitosamente.

- 4. La última prueba que vamos a realizar con el mismo token es una petición DELETE al endpoint de nuestro backend, esta petición elimina el usuario, para este proceso tenemos que enviar el id del usuario que vamos a eliminar, tal como se puede visualizar en la imagen 7.19:

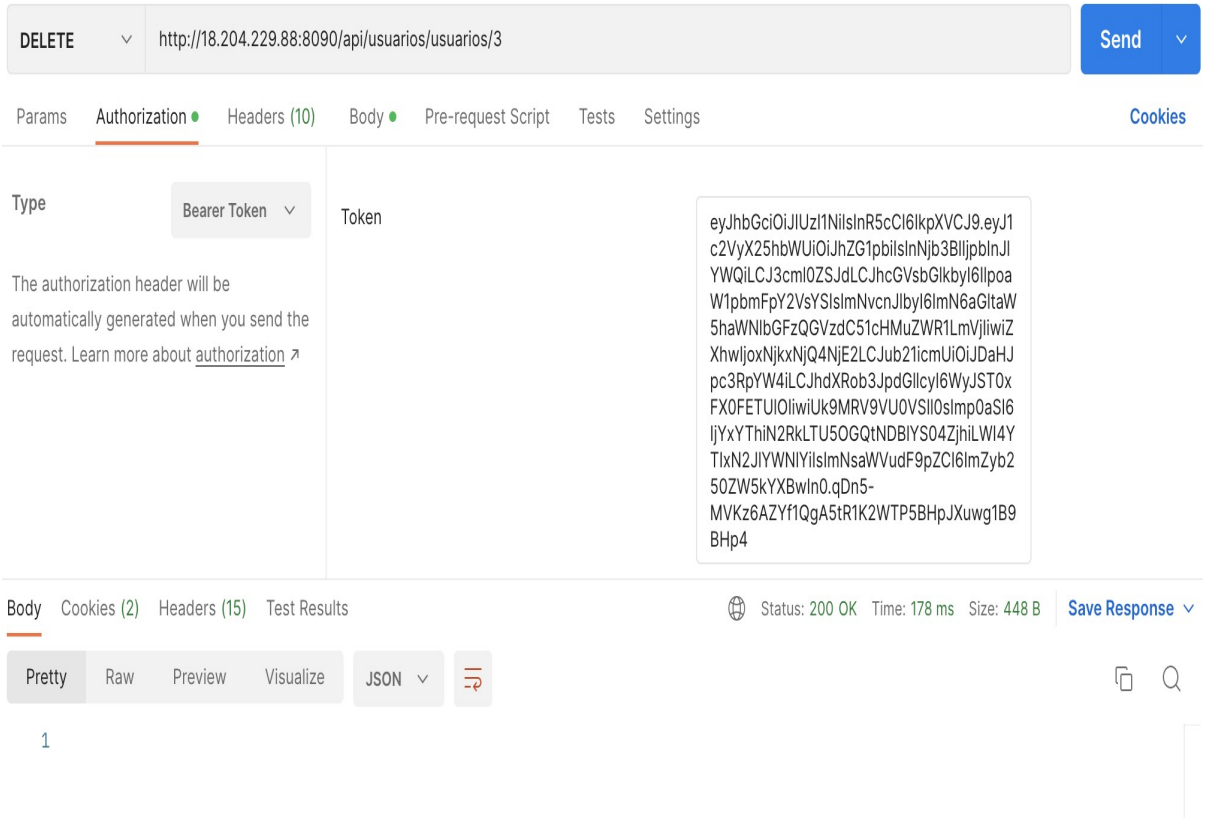



Figura 7.19: Usuario Eliminado con éxito.

## 7.5 Resultados de trazabilidad distribuida con zipkin

La implementación del software zipkin, nos permite realizar pruebas de seguimiento de trazas, para poder observar cómo se comunican los servicios, y poder saber la salud del ecosistema.

Gracias al servidor zipkin se rastrea las solicitudes que se realizan en el backend. En la siguiente imagen observamos una lista de peticiones realizadas a nuestro backend, todas las solicitudes ingresan por nuestro api gateway, tal como se puede apreciar en la imagen, el servidor zipkin captura las peticiones y nos muestra los tramos que tuvo y el tiempo de duración y cuando empezó la solicitud.

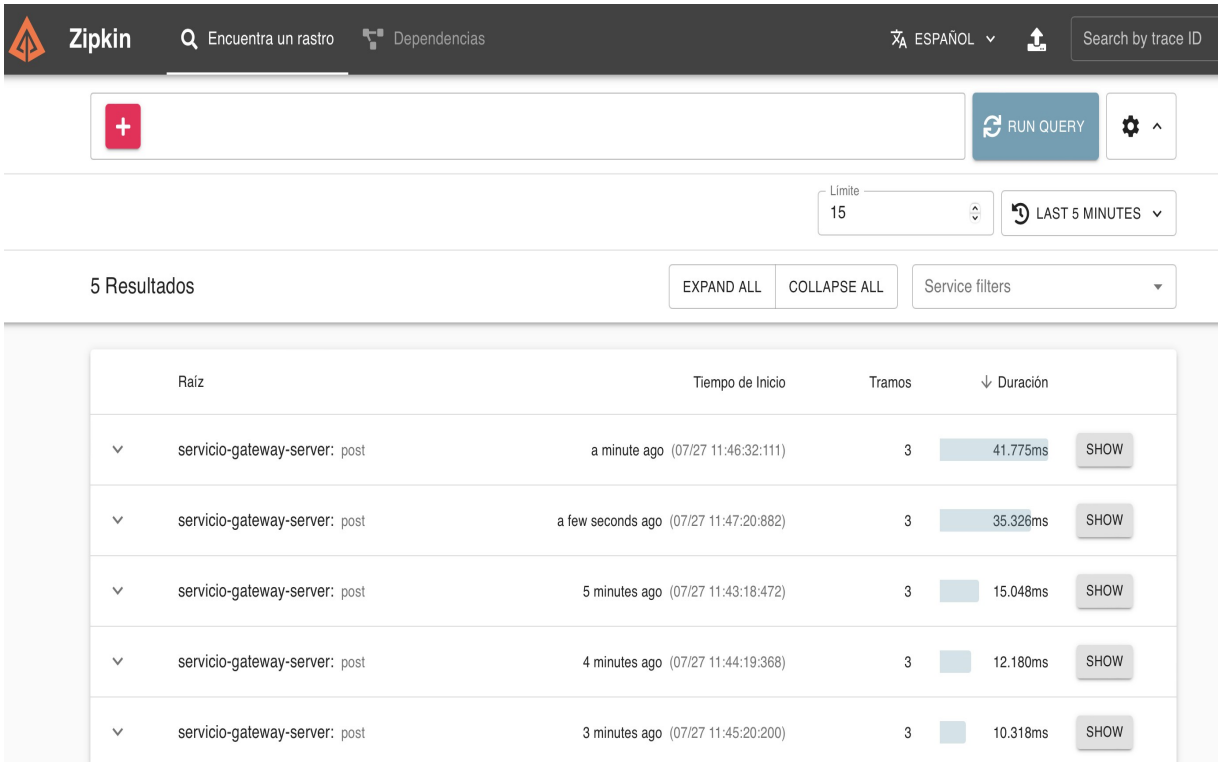



Figura 7.20: Solicitudes capturadas por zipkin.

Zipkin nos permite visualizar todas las anotaciones de las solicitudes ingresadas al backend, como vimos en la anterior imagen anterior 7.20 "Solicitudes capturadas por zipkin", son todas las peticiones que tenemos en ese momento, ahora se puede elegir una y visualizar su información, como qué tipo de método ingresa, el path al que envía la solicitud, la dirección IP a donde se envía y la dirección IP del cliente que envía, tal como se observa en la siguiente imagen.

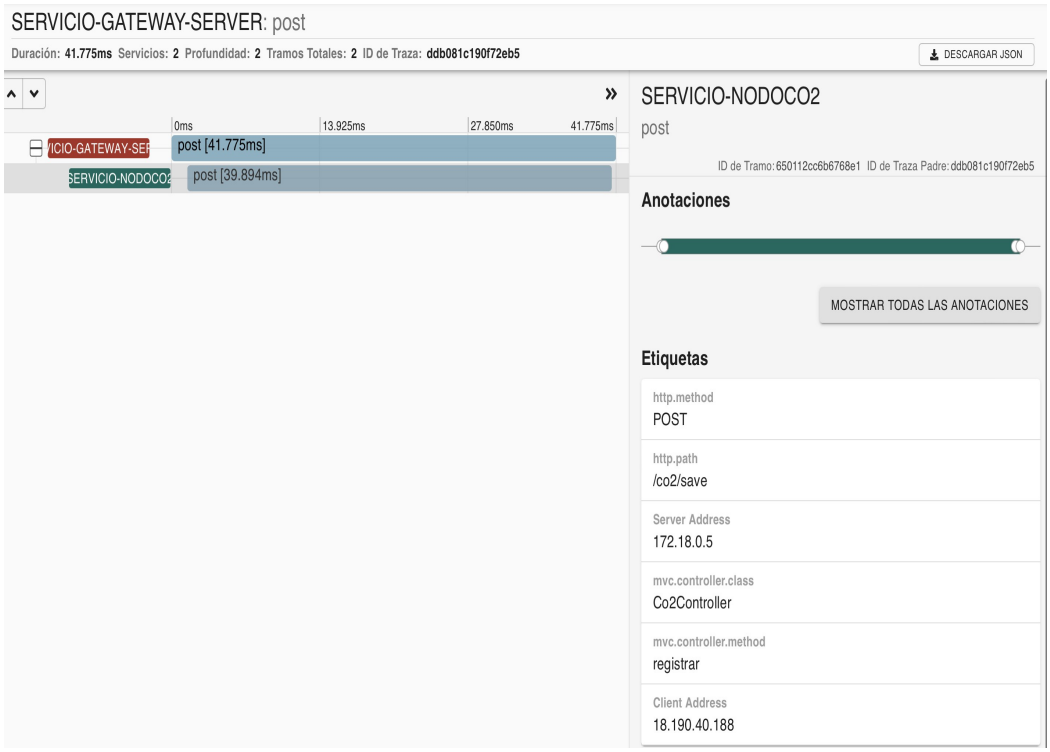



Figura 7.21: Anotaciones de la solicitud al servicio nodoco2.

### 7.5.1 Árbol de dependencias con zipkin

Zipkin nos ofrece funcionalidades que permite observar la comunicación entre microservicios, nos permite filtrar por tiempo de inicio y tiempo fin para poder observar el comportamiento del ecosistema. En la siguiente imagen se puede visualizar como el api gateway se comunica con el servicio del nodoco2, los puntos azules que se observan son las peticiones que se están enviando, en este caso son peticiones POST que envía el servidor chirpstack a nuestro api gateway del backend, ya que toda solicitud debe ingresar por una sola puerta API, así damos protección y seguridad a los microservicios.


Figura 7.22: Solicitud al servicio NodoCo2.

De acuerdo con las pruebas realizas con el cliente Postman, enviamos varias peticiones de prueba de gestión de usuario, como la de autenticación de usuario. La petición es enviada al backend, dentro de zipkin las peticiones son los puntos azules, entonces primero ingresa al API gateway la cual se comunica con el servicio requerido en este caso el servicio OAuth, el cual genera un token y devuelve la repuesta al gateway para que enviase al cliente postman, enseguida que se verifica se realiza una petición al servicio de usuarios para poder acceder a los recursos protegidos, tal como vemos en la imagen 7.23.

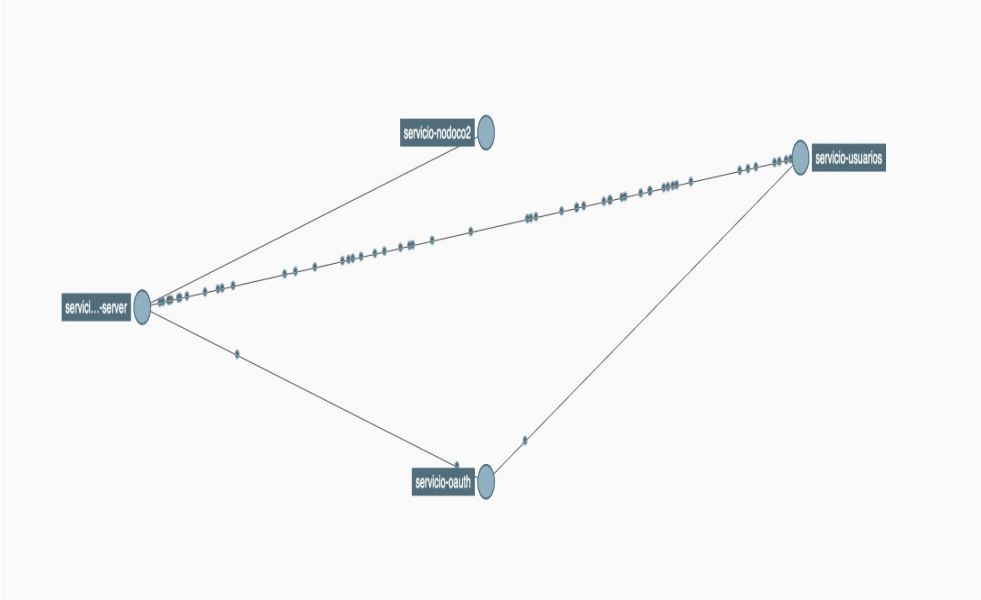
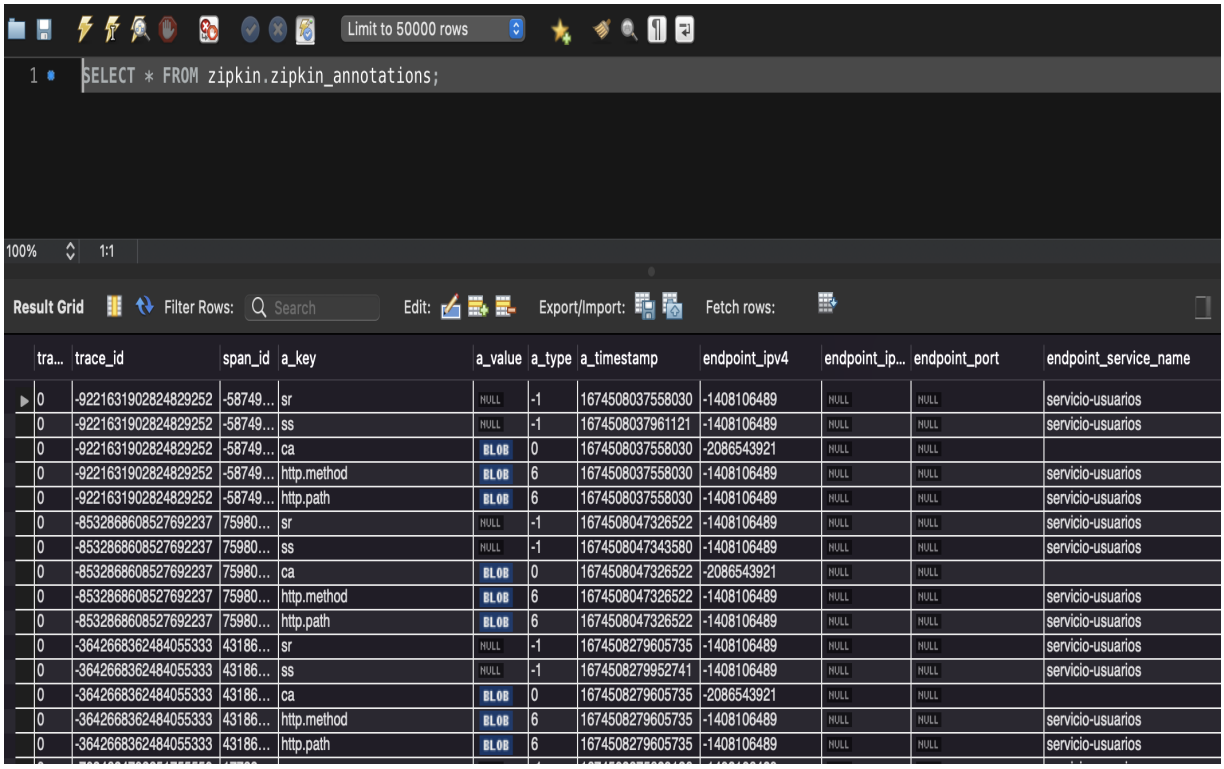



Figura 7.23: Árbol de dependencias con Zipkin.

### 7.5.2 Solicitudes almacenadas por zipkin en una base de datos MySQL

Dentro de la base de datos se puede apreciar las anotaciones que fueron almacenadas por cada solicitud, como el nombre del servicio, el id de la traza, el puerto y entre otros detalles, esto nos permite poder identificar las trazas que realiza cada servicio, tal como se aprecia en la siguiente imagen.




| tra... | trace_id | span_id | a_key | a_value | a_type | a_timestamp | endpoint_ipv4 | endpoint_ip... | endpoint_port | endpoint_service_name |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | -9221631902824829252 | -58749... | sr | NULL | -1 | 1674508037558030 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -9221631902824829252 | -58749... | ss | NULL | -1 | 1674508037961121 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -9221631902824829252 | -58749... | ca | BLOB | 0 | 1674508037558030 | -2086543921 | NULL | NULL | |
| 0 | -9221631902824829252 | -58749... | http.method | BLOB | 6 | 1674508037558030 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -9221631902824829252 | -58749... | http.path | BLOB | 6 | 1674508037558030 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -8532868608527692237 | 75980... | sr | NULL | -1 | 1674508047326522 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -8532868608527692237 | 75980... | ss | NULL | -1 | 1674508047343580 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -8532868608527692237 | 75980... | ca | BLOB | 0 | 1674508047326522 | -2086543921 | NULL | NULL | |
| 0 | -8532868608527692237 | 75980... | http.method | BLOB | 6 | 1674508047326522 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -8532868608527692237 | 75980... | http.path | BLOB | 6 | 1674508047326522 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -3642668362484055333 | 43186... | sr | NULL | -1 | 1674508279605735 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -3642668362484055333 | 43186... | ss | NULL | -1 | 1674508279952741 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -3642668362484055333 | 43186... | ca | BLOB | 0 | 1674508279605735 | -2086543921 | NULL | NULL | |
| 0 | -3642668362484055333 | 43186... | http.method | BLOB | 6 | 1674508279605735 | -1408106489 | NULL | NULL | servicio-usuarios |
| 0 | -3642668362484055333 | 43186... | http.path | BLOB | 6 | 1674508279605735 | -1408106489 | NULL | NULL | servicio-usuarios |

Figura 7.24: Solicitudes almacenadas en MySQL.

## 7.6 Sistema desplegado en el servidor del grupo de investigación GIHP4C

En la siguiente imagen se puede apreciar la creación de la máquina virtual para el despliegue de la plataforma basada en una arquitectura de microservicios, Para la creación de la maquina utilizamos el sistema operativo linux Ubuntu server versión 22, y la ip asignada de acuerdo a la administración del directo del grupo de investigación GIHP4C, configuramos la máquina virtual con la siguiente IP 172.16.26.14, tal como se aprecia en la siguiente imagen, donde se muestra las configuraciones de hardware.

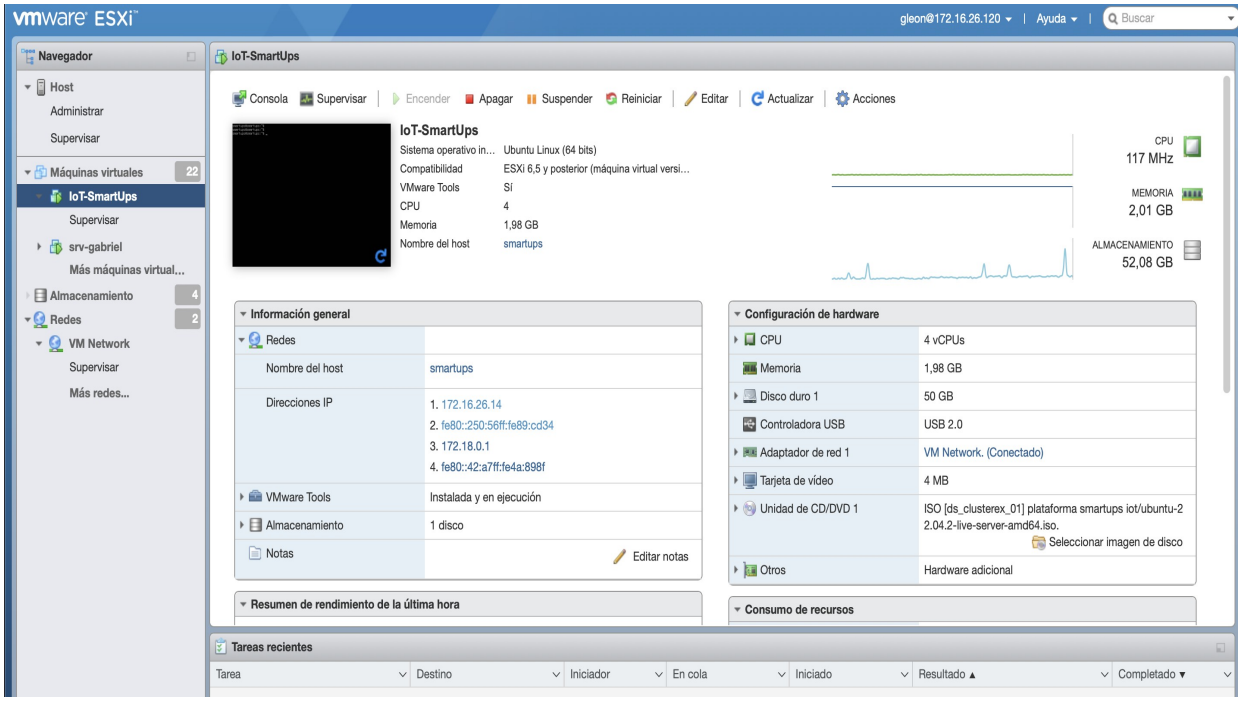



Figura 7.25: Máquina virtual creada en el servidor del grupo GIHP4C

Una vez que instalamos Ubuntu server como sistema operativo procedemos a instalar docker, para poder desplegar los microservicios, para instalar docker en nuestro linux lo hacemos con el siguiente comando:

**“sudo apt install docker-ce”**

Una vez instalado procedemos a revisar el estado del servicio docker mediante la siguiente línea de código **"sudo systemctl status docker"**. El resultado se observar en la siguiente imagen, como vemos docker se encuentra corriendo sin problemas con sus microservicios desplegados correctamente, tal como se observa los microservicios fueron registrados correctamente en eureka server, esto se aprecia en los resultados anteriores en la imagen 7.2 "Microservicios Registrados en Eureka Server".

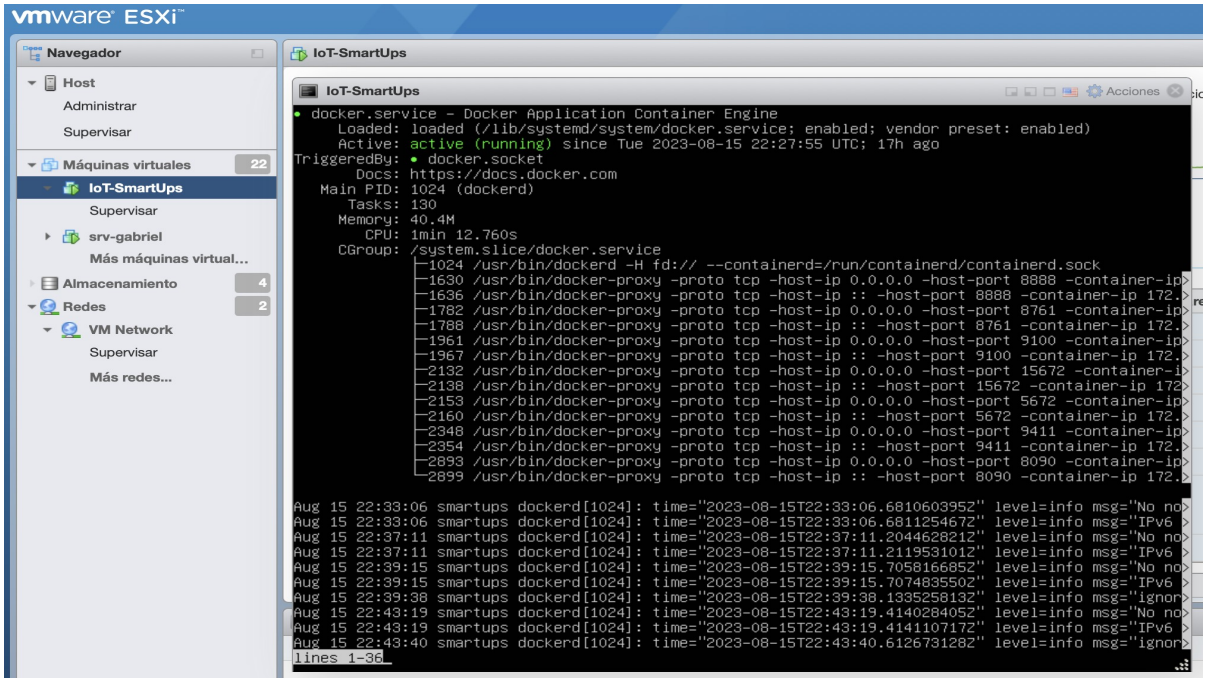



Figura 7.26: Servicio Docker corriendo en la máquina virtual.

### 7.6.1 Resultados de la Interfaz Gráfica (Frontend)

Dentro de estos resultados se observa que la plataforma está funcionando correctamente, de acuerdo con los detalles plantados, la comunicación con los microservicios es procesada con éxito en cada petición que se realiza, estos resultados los detallamos más adelante de acuerdo con cada pantalla generada.

**Página de inicio de sesión**

En la imagen 7.27 se puede observar la página inicial para poder iniciar sesión en el sistema web, la cual nos permite realizar el respectivo login a la plataforma utilizando el nombre de usuario y su contraseña, para poder tener acceso a los recursos y funcionalidades.

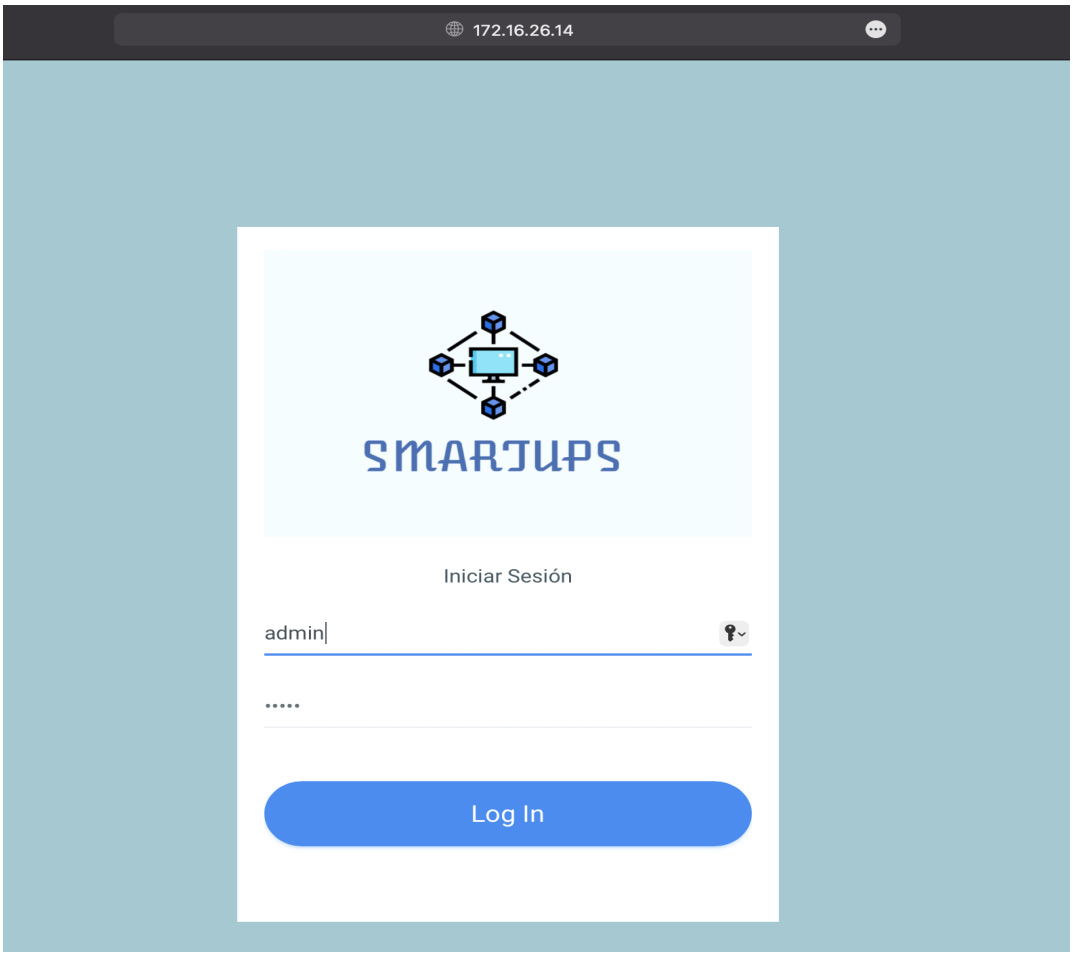



Figura 7.27: Página de inicio de sesión

**Página principal de bienvenida**

En la imagen 7.28 se puede observar la página home donde se habilitan las funcionalidades en nuestro menú dependiendo del rol de cada usuario.

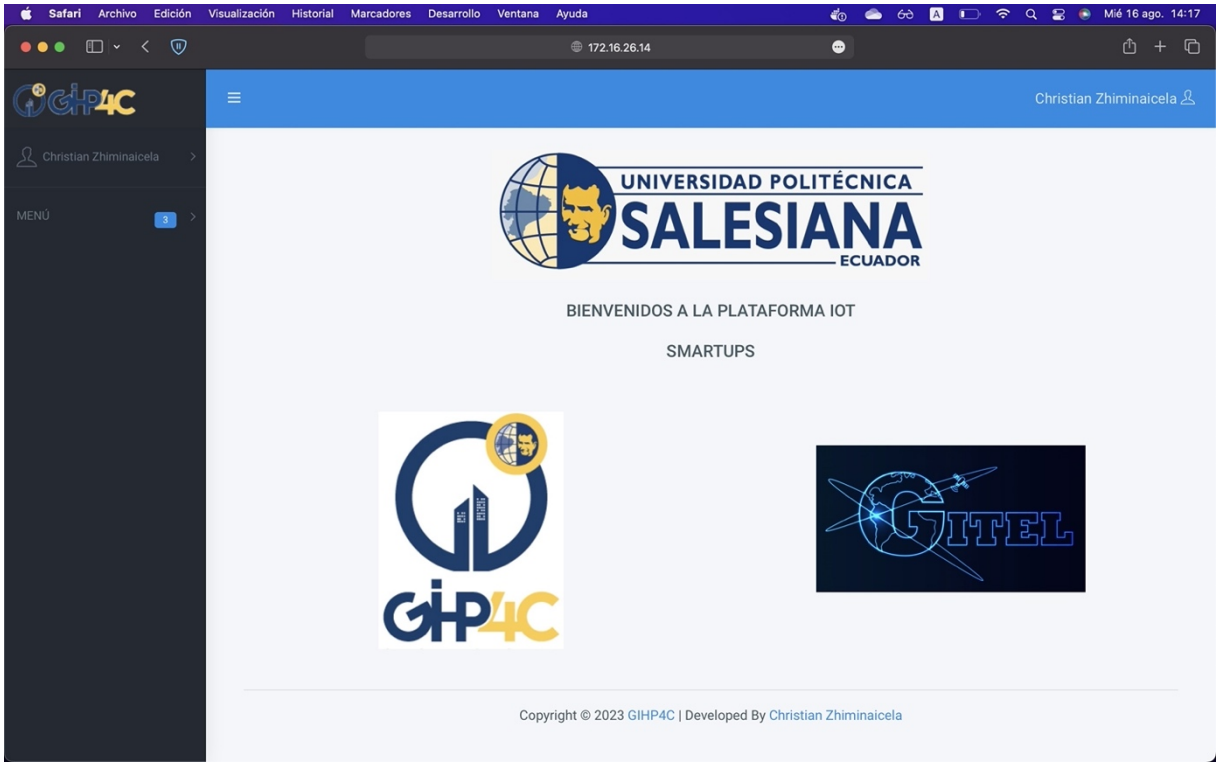



Figura 7.28: Página principal después de iniciar sesión

**Página de Gestión de Usuarios**

En la imagen 7.29 se puede observar la página de usuarios donde se realiza las siguientes acciones, eliminar, actualizar, crear y listar los usuarios. Esta funcionalidad está habilitada solo para usuarios administradores.

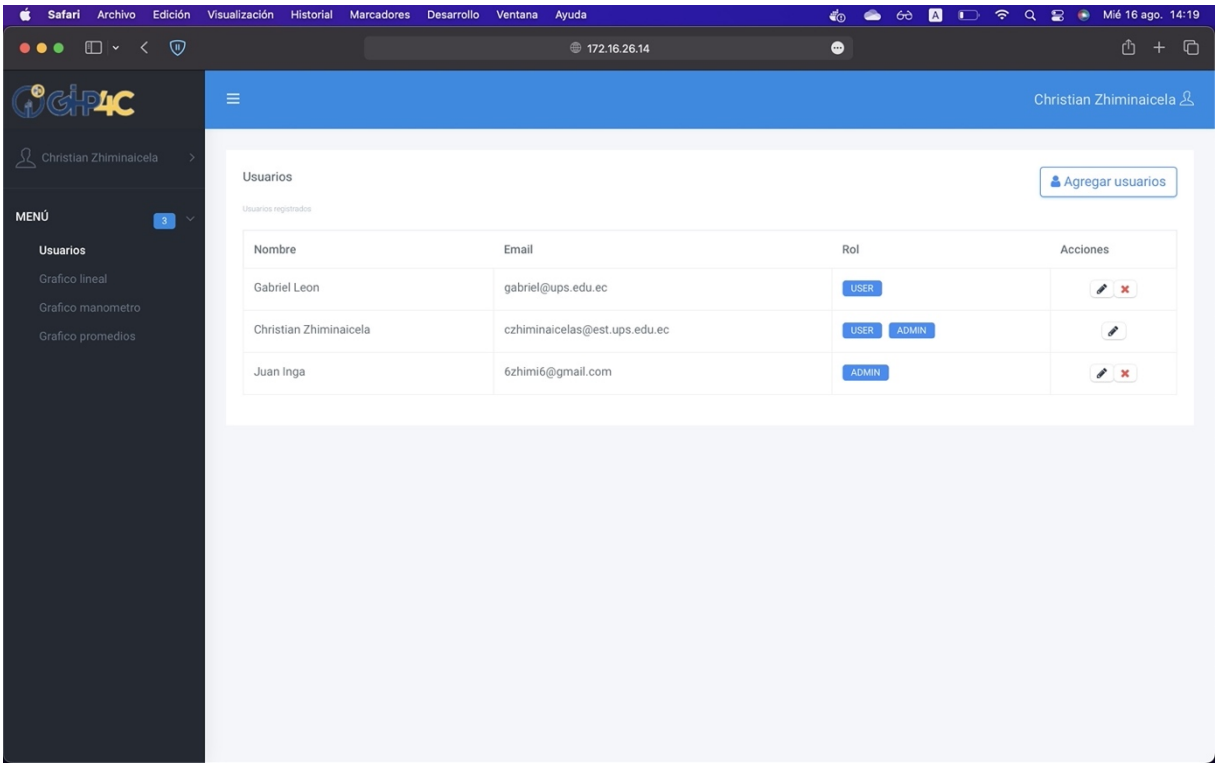



Figura 7.29: Página de Gestión de Usuarios

**Registro de nuevo usuario**

En la imagen 7.30 se puede mirar la página en donde se tiene que ingresar toda la información del nuevo usuario para poder registrar en el sistema.

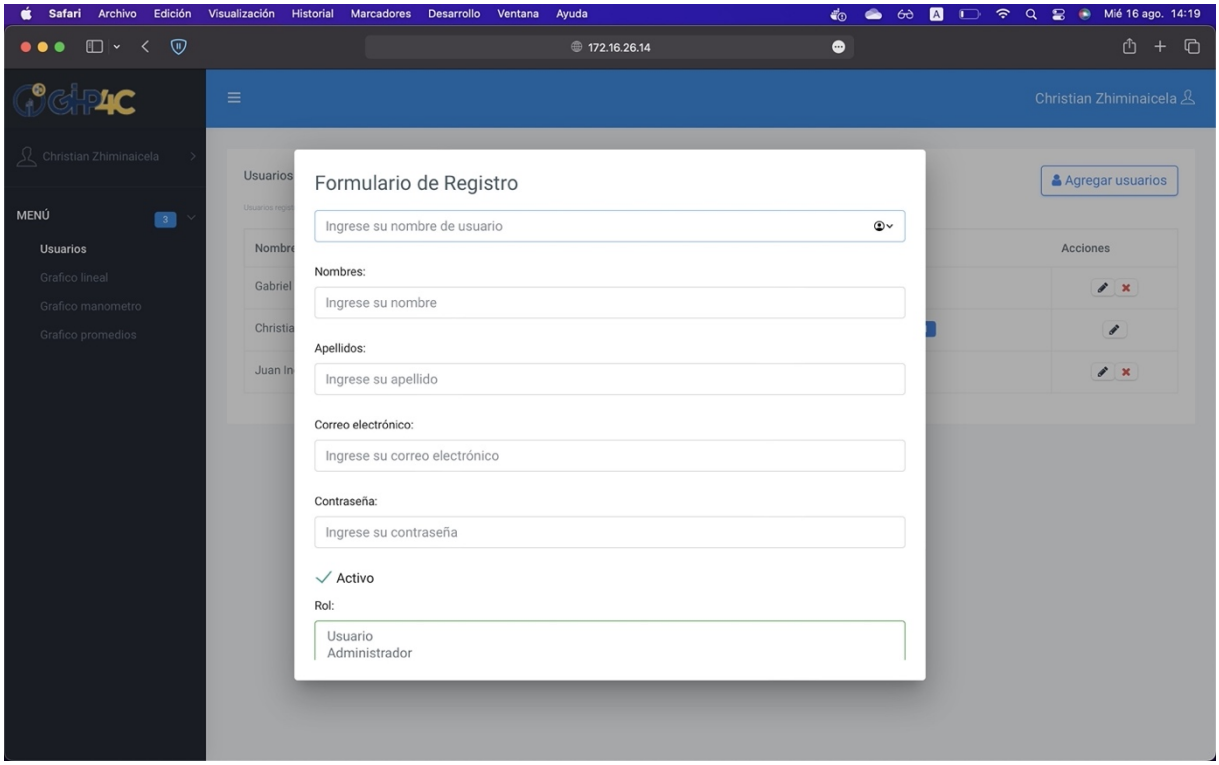



Figura 7.30: Formulario Registro Usuario.

### Gráfica Lineal de CO2

En la imagen 7.31 nos muestra la página que nos permite visualizar la gráfica lineal del Co2, seleccionando la fecha y el nodo. Además, se puede observar un manómetro que nos muestra el porcentaje del Co2, eligiendo un punto en la gráfica obtenida.

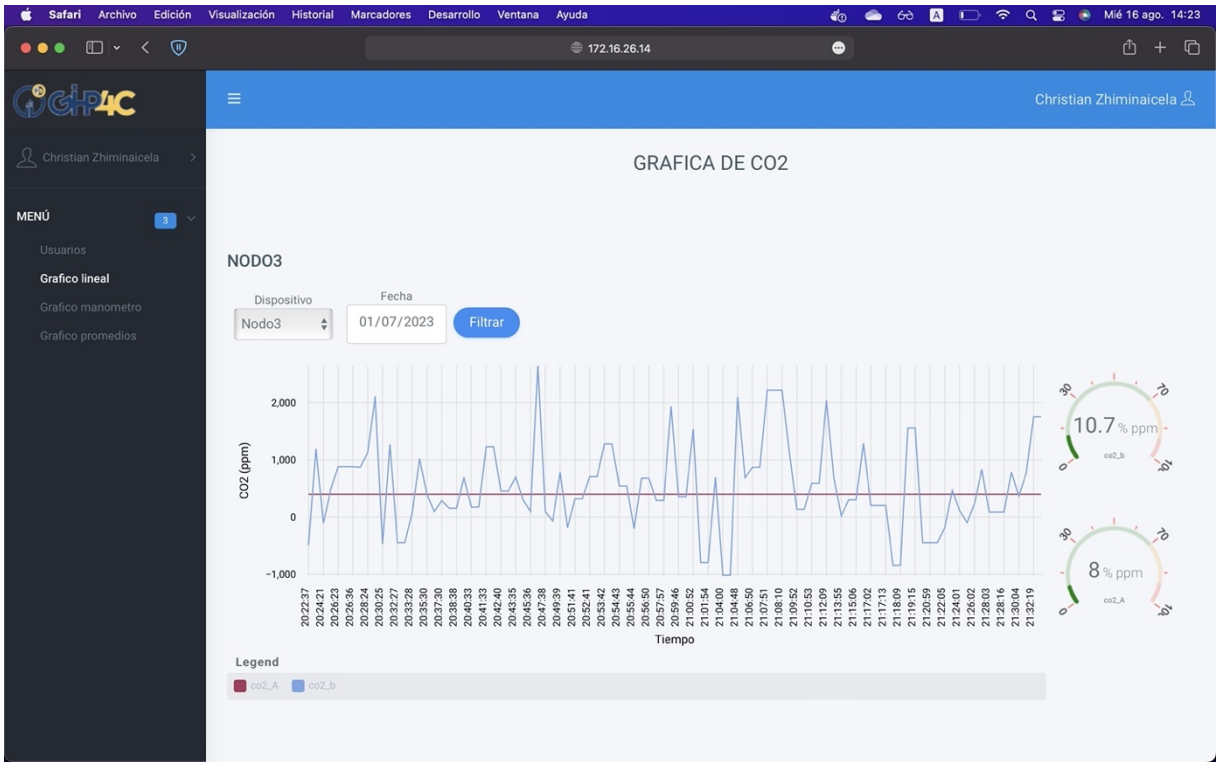



Figura 7.31: Interfaz de la gráfica lineal.

### Gráfica en Manómetro de Co2

Como se puede observar en la imagen 7.32, se puede obtener un promedio general del Co2.A y del Co2.B en ppm eligiendo la fecha y el nodo.

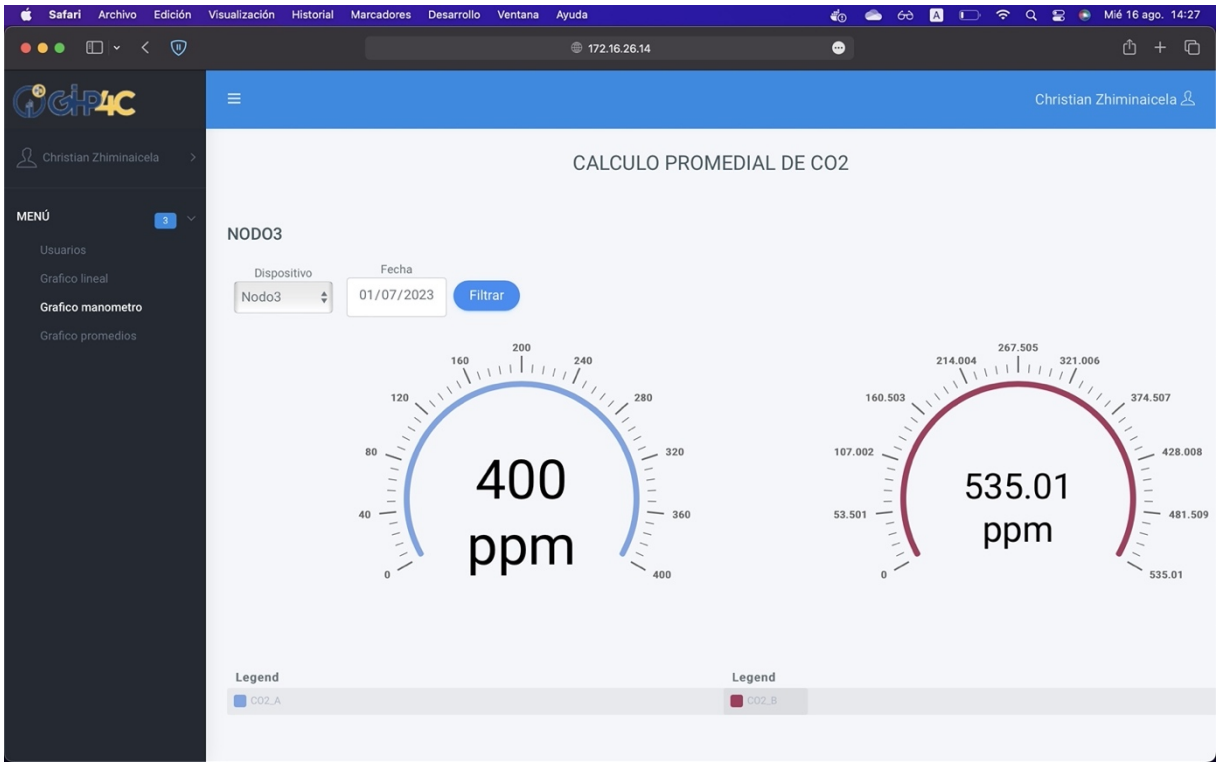



Figura 7.32: Interfaz de la gráfica manómetro.

### Gráfica del cálculo promedial por horas

En la imagen 7.33, se puede visualizar la gráfica obtenida del **NODO 3**, con sus respectivos promedios generales obtenidos por cada hora, se puede seleccionar el día del cual queremos obtener el promedio y el nombre del nodo que deseamos visualizar.

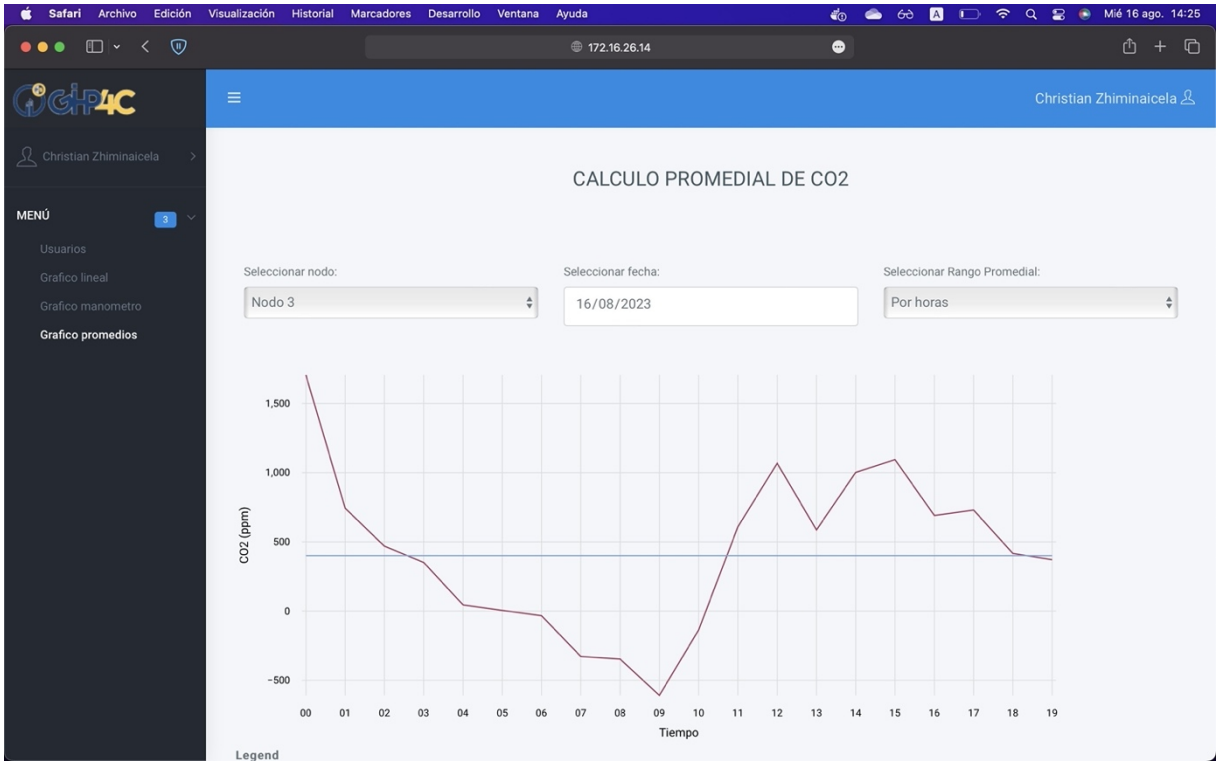



Figura 7.33: Gráfica de promedios por hora

### Gráfica del cálculo promedial por días

La siguiente imagen 7.34 nos permite visualizar la gráfica con los promedios diarios de cada nodo seleccionado, en este caso seleccionamos el **NODO 3** y depende de la fecha que se elige, también se puede seleccionar un punto dentro de la gráfica y ver los valores obtenidos del co2A y del co2B.

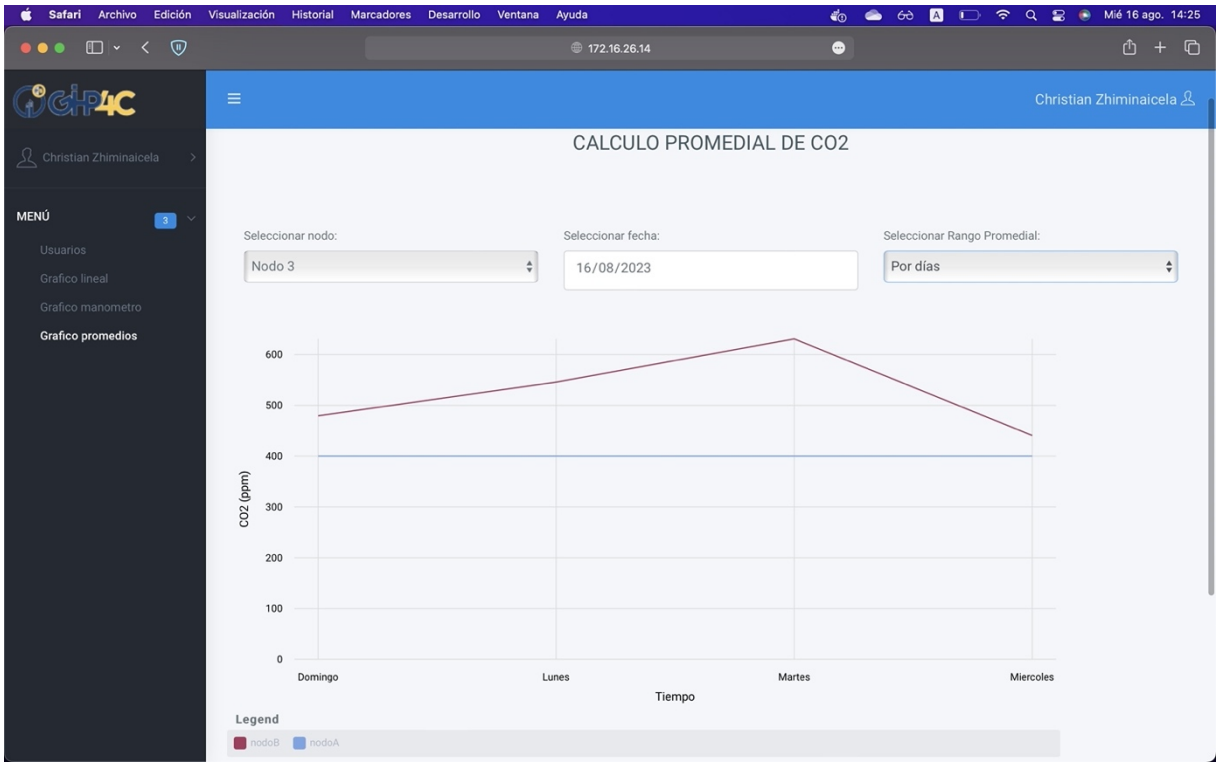



Figura 7.34: Promedio diario del Nodo 3.

### Gráfica del cálculo promedial por semanas

La siguiente imagen nos permite visualizar la gráfica con los promedios semanales de cada nodo seleccionado, en este caso nos muestra los resultados del **NODO 3** del mes de mayo, nos muestra el promedio del co2A y el co2B de cada semana dependiendo del mes elegido, tal como se ve en la imagen 7.35

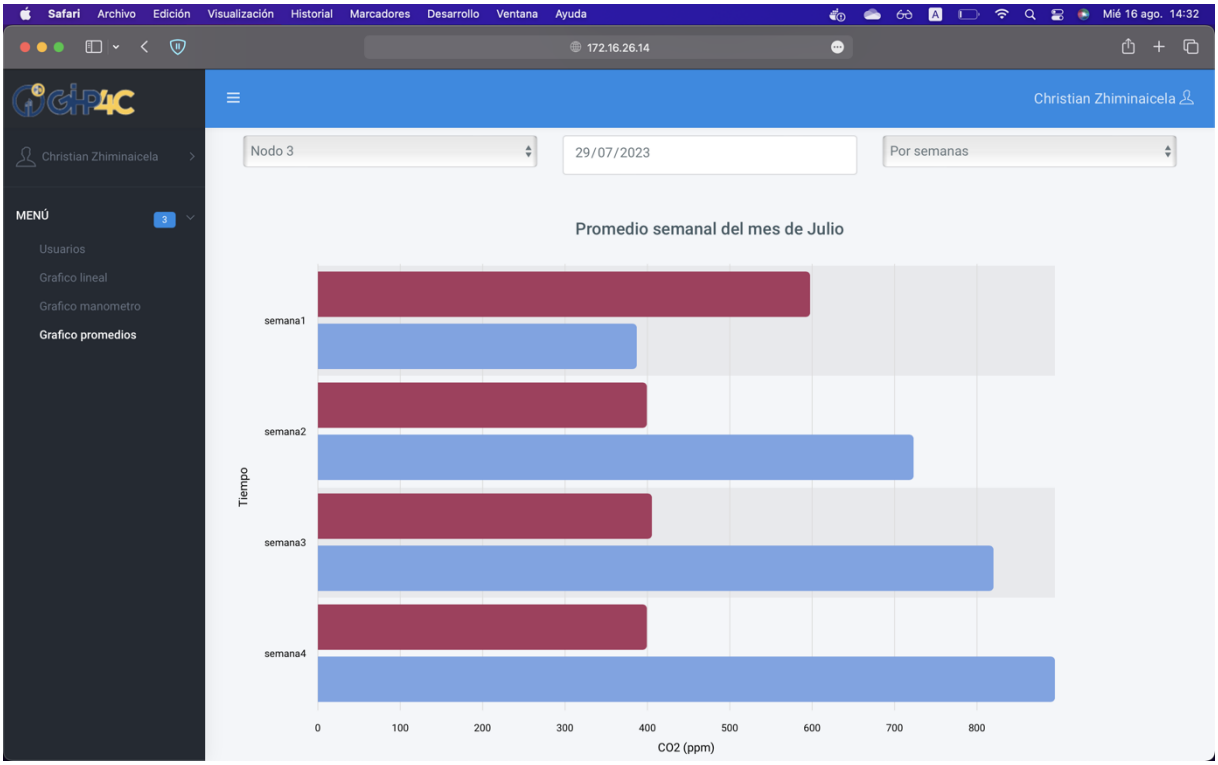



Figura 7.35: Promedio semanal del nodo 3.

### Gráfica del cálculo promedial por meses

La siguiente imagen 7.36 nos permite visualizar la gráfica con los promedios mensuales del **NODO 3**, nos muestra el promedio del co2A y co2B de cada mes del año. Si deseamos visualizar las gráficas de otro nodo solo seleccionamos el número del nodo y se genera automáticamente.

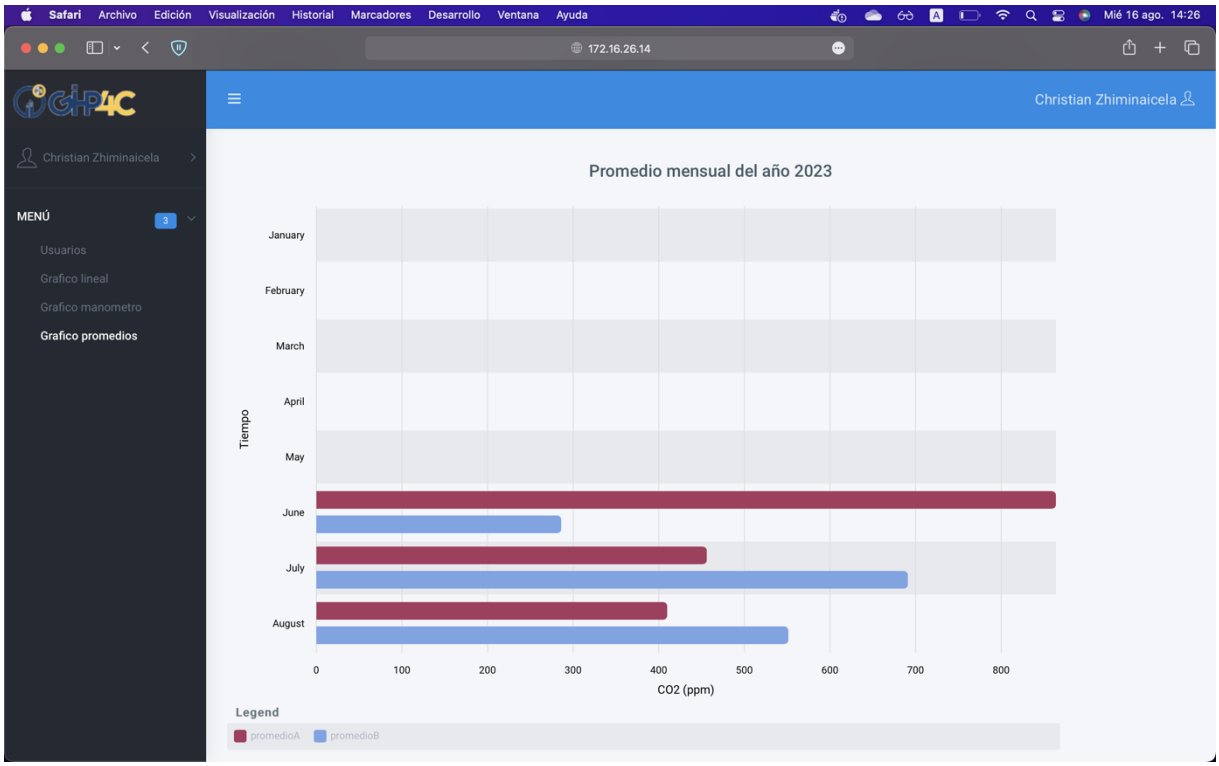



Figura 7.36: Promedio mensual del nodo 3.

# Capítulo VIII

# Cronograma

| Nombre de la tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| **Proyecto** | **769 horas** | **04/04/2022** | **00/00/0000** |
| **OE. 1** | **66 horas** | **04/04/2022** | **18/05/22** |
| ACT 1. Investigar conceptos relacionados a Cloud Computing. | 22 horas | 04/04/2020 | 15/04/22 |
| ACT 2. Investigar sobre la arquitectura de microservicios. | 22 horas | 18/04/22 | 05/05/22 |
| ACT 3. Estudiar la implementación de aplicaciones basadas en microservicios. | 22 horas | 10/05/22 | 18/05/22 |
| **OE. 2** | **100 horas** | **23/05/22** | **30/06/22** |
| ACT 1. Reuniones para la toma de requerimientos. | 22 horas | 23/05/22 | 31/05/22 |
| ACT 2. Análisis y Diseño de requerimientos. | 22 horas | 02/06/22 | 10/06/22 |
| ACT 3. Análisis y Diseño de la arquitectura propuesta. | 28 horas | 13/06/22 | 21/06/22 |
| ACT 4. Diseño de prototipos de la plataforma web. | 28 horas | 22/06/22 | 30/06/22 |
| **OE. 3** | **233 horas** | **02/08/22** | **13/01/23** |
| ACT 1. Diseño y creación de las bases de datos. | 63 horas | 02/08/22 | 20/07/22 |
| ACT 2. Desarrollo de los Microservicios. | 88 horas | 23/07/22 | 12/09/22 |
| ACT 3. Desarrollo de interfaz para el usuario final. | 82 horas | 03/10/22 | 13/01/23 |
| **OE. 4** | **170 horas** | **25/01/23** | **30/03/23** |
| ACT 1. Despliegue de la plataforma web para los servicios IoT. | 90 horas | 25/01/23 | 02/03/23 |

| | | | |
|---|---|---|---|
| ACT 2. Pruebas de interconexión entre dispositivos IoT y plataforma web. | 80 horas | 06/03/23 | 30/03/23 |
| **OE. 5** | **200 horas** | **15/05/23** | **17/07/23** |
| ACT 1. Escritura Académica. | 180 horas | 15/05/23 | 03/07/23 |
| ACT 2. Revisiones y Correcciones. | 20 horas | 17/07/23 | 10/08/23 |

# Capítulo IX

# Presupuesto

| DENOMINACIÓN | CANTIDAD | COSTO UNITARIO | COSTO TOTAL |
|---|---|---|---|
| | unidades | dólares | dólares |
| 1. Tecnológico | | | |
| Computador Portátil | 2 | 1 200,00 | 2 400,00 |
| Cursos Online | 3 | 25,00 | 75,00 |
| Libros Digitales | 2 | 50,00 | 100,00 |
| 2. Servicios | | | |
| Servicio de Transporte | 1 | 200,00 | 200,00 |
| 3. Personal | | | |
| Estudiante/ Desarrollador | 2 | 6 000,00 | 1 2000,00 |
| Asesoría especializada | 15 | 20 | 300,00 |
| 4. Otros | | | |
| Imprevistos | 1 | 250 | 250,00 |
| Total | 26 | **7 745,00 $** | **15 325,00 $** |

# Capítulo X

# Conclusiones

En este proyecto se han alcanzado resultados satisfactorios, logrando desarrollar una solución para IoT para el monitoreo de CO2 en la Universidad Politécnica Salesiana, consideramos que la solución implementada es ligera, robusta y escalable. El backend basado en una arquitectura de microservicios permite incorporar nuevos servicios a nuestro servidor, gracias a la implementación de nuestro servidor de configuraciones centralizado con github, esto permite la incorporación de nuevos servicios dentro de nuestro ecosistema de microservicios. La implementación de docker permite realizar un despliegue continuo cada vez que tengamos nuevas actualizaciones, sin tener que parar todos los servicios.

El desarrollo de nuestro frontend nos permite visualizar e interpretar la información de los sensores de co2 con más facilidad. La arquitectura del backend basada en microservicio fue diseñada pensando en ser extensible, por lo que es posible introducir nuevos microservicios para soportar más funcionalidades en estos componentes de acuerdo con las necesidades del grupo de Investigación en “Cloud Computing, Smart Cities & High Performance Computing (GIHP4C)” y por el grupo de Investigación en “Telecomunicaciones y Telemática (GITEL)” de la Universidad Politécnica Salesiana.

Spring boot es una herramienta principal para la creación y el mantenimiento de sistemas con arquitectura basada en microservicios, gracias a su gran equipo de desarrolladores que siguen implementando y actualizando nuevas funcionalidades para mejorar la programación de microservicios. “Las Arquitecturas basadas en microservicios han llegado para cambiar todo el paradigma sobre el desarrollo de sistemas, ofreciendo una solución práctica y más eficiente, gracias a los avances de los frameworks y entornos de desarrollo, los cuales, permiten reducir considerablemente el tiempo y costo en la producción de sistemas”

# Capítulo XI

# Recomendaciones

Es importante contar con arquitecturas de microservicios, gracias a su estructura se desarrolló nuevos servicios e implementar dentro de la misma arquitectura sin mucha complejidad y sin parar sus funciones. Para el desarrollo de nuevos sistemas con arquitecturas de microservicios se recomienda utilizar siempre un servidor de configuraciones, ya que este servidor nos permite centralizar todas las configuraciones, para conectar a las bases de datos y sobre todo para la comunicación entre microservicios, teniendo así un sistema con alta disponibilidad y sin tener que bajar cada microservicio solo para modificar los cambios que se realizaron dentro de las configuraciones. Otro aspecto importante es la seguridad, gracias a la implementación de spring security con oauth y jwt, se puede restringir el acceso a los microservicios y sus recursos de acuerdo con el manejo de roles de usuario que implementamos dentro de nuestro sistema. En futuros proyectos se debe considerar utilizar otros componentes y protocolos de comunicación como LoRaWAN, MQTT, CoAP o XMPP, estos nos pueden facilitar para tener un mejor rendimiento y estabilidad en la comunicación de dispositivos IoT.

# Referencias bibliográficas


[1] A. M. Cornejo Orellana and C. F. Díaz Escalante. Análisis, diseño e implementación de cloud computing para una red.

[2] Isaac Odun-Ayo, M Ananya, Frank Agono, and Rowland Goddy-Worlu. Cloud computing architecture: A critical analysis. In *2018 18th international conference on computational science and applications (ICCSA)*, pages 1–7. IEEE, 2018.

[3] Mohammad Aazam, Sherali Zeadally, and Khaled A Harras. Fog computing architecture, evaluation, and future research directions. *IEEE Communications Magazine*, 56(5):46–52, 2018.

[4] Dinesh Kumar Saini, Krishan Kumar, and Punit Gupta. Security issues in iot and cloud computing service models with suggested solutions. *Security and Communication Networks*, 2022, 2022.

[5] Sergio Sánchez Prado. Cloud computing: fundamentos y despliegue de un servicio en la nube. B.S. thesis, 2021.

[6] Viudez Corroto González Madraño, López Collado. An open cloud computing interface (occi) management console for the opennebula toolkit. 2010.

[7] H. Maharwal N. Jain y A. Dadhich R. Kumar, K. Jain. Open-source infrastructure as a service cloud computing platform. proceedings of the international journal of advancement in engineering technology. *Apache cloudstack*, pages ágina–111, 2014.

[8] Red Hat. Inc. «openstack documentation,» 2020. *[En línea] . Available: https://www.redhat.com/es/topics/openstack/*.

[9] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha, and Manuel Mazzara. Micro-frontends: application of microservices to web front-ends. *J. Internet Serv. Inf. Secur.*, 10(2):49–66, 2020.

[10] Santiago Ramírez Pérez et al. Estudio del framework spring, spring boot y microservicios. 2020.

[11] Hanin M Abdullah and Ahmed M Zeki. Frontend and backend web technologies in social networking sites: Facebook as an example. In *2014 3rd international conference on advanced computer science applications and technologies*, pages 85–89. IEEE, 2014.

[12] . Overview of the angular framework: pros and cons. , page 33, 2020.

[13] Docker. Docker doc. 2022.

[14] Andrés Nebel. Arquitectura de microservicios para plataformas de integración. 2019.

[15] Daniel López, Edgar Maya, et al. Arquitectura de software basada en microservicios para desarrollo de aplicaciones web. 2017.

[16] Pooja Sharma. Microservicios vs api: sepa qué es lo mejor para su negocio. 2022.

[17] Sonia Mora González. Entendiendo el internet de las cosas. *Investiga. TEC*, (24):ágina–22, 2015.

[18] MediaDSCI. Iot technology in india. *blog*, 2020.

[19] A. Heidari P. Allahverdizadeh J. Jamali, B. Bahrami and F. Norouzi. Towards the internet of things. *Springer International Publishing, first ed.*, 2020.

[20] L. Khalid. Software architecture for business. *Springer International Publishing, first ed.*, 2020.

[21] Sacoto-Cabrera, E. J., Castillo, I., Pauta, W., Trelles, P., Tamaríz, P., & Guambaña, L. (2022, November). Smart-Water: Digital Transformation of Urban Water Measurement. In 2022 IEEE ANDESCON (pp. 1-6). IEEE.

[22] L. del Valle Hernández. Programar fácil. *[En línea]. Available: https://programarfacil.com/podcast/proyectos-iot-con-arduino*, 2021.

[23] Sacoto Cabrera, E. J., Palaguachi, S., León-Paredes, G. A., Gallegos-Segovia, P. L., & Bravo-Quezada, O. G. (2020, October). Industrial communication based on mqtt and modbus communication applied in a meteorological network. In The International Conference on Advances in Emerging Trends and Technologies (pp. 29-41). Cham: Springer International Publishing.

[24] Vimos, V., & Cabrera, E. J. S. (2018). Results of the implementation of a sensor network based on Arduino devices and multiplatform applications using the standard OPC UA. IEEE Latin America Transactions, 16(9), 2496-2502.

[25] Loza Peralta Christian Víctor. Arquitectura de software basada en microservicios para el uso en dispositivos de internet de las cosas. thesis, 2020.

[26] Redacción APD. Metodología scrum. *[En línea] . Available: https://www.apd.es/metodologia-scrum-que-es/*, 2019.

[27] Antonio Jesús González García. Iot: Dispositivos, tecnologías de transporte y aplicaciones. 2017.

[28] Isuru Jayakantha. Microservicios - registro y descubrimiento de servicios con netflix eureka. *[En línea] . Available: https://medium.com/@ijayakantha/microservices-service- registration-and-discovery-with-netflix-eureka-9a2aa729da96*, 2019.

[29] Rodríguez Moreno, Edward Stiven, and Víctor Felipe López Ordoñez. "Diseño e

implementación de un sistema inteligente para un edificio mediante IOT utilizando el protocolo de comunicación LORAWAN." (2017).

[30] Sarmiento, Alexander Eslava. *Logística de transporte de mercancías en contenedores marítimos*. Ediciones de la U, 2019.

[31] Cadavid, Andrés Navarro, Juan Daniel Fernández Martínez, and Jonathan Morales Vélez. "Revisión de metodologías ágiles para el desarrollo de software." *Prospectiva* 11.2 (2013): 30-39.

[32] Lavado Haro, Julio Cesar. "Desarrollo E Implementación De Una Aplicación Móvil, Basada En La Metodología Scrum, Para La Mejora Del Proceso De Captura De Datos De La Encuesta Nacional De Calidad A Mypes Del Instituto Nacional De Estadística E Informática." (2018).

[33] Maida, Esteban Gabriel, and Julián Pacienzia. "Metodologías de desarrollo de software." (2015).

[34] Díaz Chávez, Patricia Milagros, and Raal Pablo Rojas Arroyo. "Propuesta de una arquitectura empresarial para una consultora de software."

[35] Chaves, M. A. (2005). La ingeniería de requerimientos y su importancia en el desarrollo de proyectos de software. InterSedes: Revista de las Sedes Regionales, 6(10), 1-13

[36] IEEE Computer Society, 2004 http://www.cc.uah.es/drg/b/HispaSWEBOK.Borrador.pdf

[37] Sommerville, I. (2005). Requerimientos del software. Ingeniería del software, 7a ed., PEARSON EDUCACIÓN, Madrid, SPA, 109-110.