UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

APLICACIÓN BASADA EN ARQUITECTURA DE MICROSERVICIOS

APPLICATION BASED ON MICROSERVICES

Raúl Diego Navarro y Ricardo Daniel Cabrera Lozada

rauldieg@ucm.es y rcabre01@ucm.es

Dirigido por: Eduardo Huedo Cuesta

Curso Académico: 2019/2020

RESUMEN

El principal objetivo de este trabajo es implementar una arquitectura de microservicios con una malla de servicios (*service mesh*). Hablaremos de las ventajas y desventajas de una arquitectura de microservicios frente a una monolítica, y las ventajas que ofrecen las mallas de servicio a nivel seguridad, conectividad, observabilidad y control.

Los servicios de este proyecto han sido implementados con los diferentes lenguajes que predominan hoy en día, como Python, Java, PHP o Spring Boot, entre otros. Además, contienen varias APIs REST y una base de datos SQL. En definitiva, tratamos los principales paradigmas de los desarrolladores DevOps.

Estos servicios van a ser desplegados en Docker y Kubernetes y por último se incluirá Istio, donde profundizaremos más, debido a que es la plataforma que nos posibilita el desarrollo de la capa de servicio.

ABSTRACT

The main objective of this project is to develop a microservices architecture within a service mesh. We introduce the advantages and disadvantages of a microservices architecture compared to a monolithic one, and the advantages that a service mesh offers in terms of security, connectivity, observability and control issues.

The services have been implemented with different languages, like PHP, Java, Spring Boot or Python, among others. They also contain several REST APIs and a SQL database. In short, we treat the main paradigms of DevOps developers.

These services are going to be deployed in Docker and Kubernetes, and finally Istio will be included, where we will delve deeper, because it is the platform that allows us to develop the service mesh.

PALABRAS CLAVE

- Microservicios
- Docker
- Kubernetes
- Istio
- Malla de servicio
- Clúster
- Contenedores

KEY-WORDS

- Microservices
- Docker
- Kubernetes
- Istio
- Service mesh
- Cluster
- Container

INDICE

INDICE DE FIGURAS

INDICE DE TABLAS

1. INTRODUCCIÓN

Los desarrollos de sistemas de gran envergadura y complejos normalmente se realizaban basándose en la arquitectura monolítica, en la cual mantener y escalar es muy complicado. En la actualidad se observa como las empresas están decantándose por arquitecturas de microservicios, ya que les otorga muchos beneficios como son estandarización, escalabilidad, mantenimiento y agilidad.

Ejecutar microservicios no es complicado, porque tenemos muchas tecnologías disponibles para ayudar a su implementación, como Spring Boot, Django, Lumen framework, todas ellas pueden ayudar a solventar los problemas relacionados con su construcción.

Desplegar este tipo de arquitecturas no es muy complejo, ya que actualmente existen softwares que ayudan con el proceso como son PodMan, Docker, Kubernetes, entre otros.

En este punto se obtienen problemas con arquitectura de microservicios como son la de poder controlar todos los microservicios, las comunicaciones y la trazabilidad. Para solventar este problema se implementa una malla de servicios (service mesh) que es una infraestructura encargada de controlar toda la comunicación y seguridad de los microservicios. De esta manera se reduce la complejidad que se tiene al implementar este tipo de arquitectura.

Para desarrollar la malla de servicios, se utiliza Istio, que es una plataforma de código abierto la cual nos permite gestionar la comunicación, la seguridad y los indicadores de los microservicios.

1.1. MOTIVACIÓN

La motivación de realizar este proyecto es debido a la creciente demanda de las organizaciones en el uso de arquitectura basada en microservicios, por lo que queremos observar los beneficios y desventajas. De esta manera crearemos una aplicación para poner en marcha la mayor parte de las funcionalidades y conceptos.

1.2. OBJETIVOS PRINCIPALES

OBJETIVO GENERAL

El objetivo principal del proyecto es implementar una aplicación basada en microservicios desde cero, usando una malla de servicio.

OBJETIVOS ESPECÍFICOS

- Documentar todo el proceso llevado a cabo. Lo que nos va a permitir ver los beneficios y desventajas de utilizar este tipo de arquitectura, así como de la malla de servicio.
- Aprender a usar todas estas nuevas tecnologías planteadas como son Spring Boot, Python, Kubernetes e Istio.

1.3. PLANIFICACIÓN

- En la primera fase del proyecto el objetivo es aprender las tecnologías Spring Boot, PHP, Python, Docker, Kubernetes e Istio.
- En la segunda fase, se diseñarán los microservicios que se crean necesarios para comenzar la implementación.
- En la tercera fase, se desarrollarán los microservicios en los distintos leguajes de programación.
- En la cuarta fase, se desplegarán los microservicios mediante Kubernetes.
- En la quinta fase, se añade Istio al proyecto.
- En la sexta fase, se redacta la memoria y se muestra los resultados obtenidos.

2. INTRODUCTION

The developments of complex systems at large scale have been traditionally based on a monolithic architecture, which it is not easy to understand or maintain. Nowadays, organizations are more interested on a microservice architecture due to its benefits in terms of standardization, scalability, maintenance and agility.

The implementation of microservices is not complicated because there are many available technologies like Spring Boot, Django, Lumen framework to make it easier. All of them can help to solve problems related to its construction.

Deploying this type of architectures is not complicated because there is software like PodMan, Docker, Kubernetes, among others that can help with the process.

At this point, problems with microservices architecture can appear, such as the lack of control over all the microservices, communications and the traceability. To solve these problems, we can implement a service mesh, which is an infrastructure responsible for controlling the communication and security of the microservices. In this way, the complexity of implementing this type of architecture decrease.

To develop the service mesh, Istio is the best tool. It is an open source platform that allows us to manage the communication, security and microservices' indicators.

2.1. MOTIVATION

The motivation to carry out this project is due to the increasing demand of organizations in the use of microservice architecture, so we want to observe the benefits and disadvantages. In this way we will create an application to implement most of the functionalities and concepts.

2.2. MAIN OBJETIVES

GENERAL OBJECTIVE

The general objective is to implement an application based on microservices from scratch, by using a service mesh.

SPECIFIC OBJECTIVES

- To document all the process, which will allow us to evaluate the advantages and disadvantages of using this architecture as well as the service mesh.
- To learn how to use all these new technologies, like Spring Boot, Python, Kybernetes and Istio.

2.3. PLANNING

- In the first phase of the project, the objective is to learn about Spring Boot, PHP, Python, Docker, Kubernetes and Istio.
- In the second phase, we will design the necessary microservices for begin the implementation.
- In the third phase, we will design microservices in the different programming languages.
- In the fourth phase, we will deploy the microservices through Kubernetes.
- In the fifth phase, we will add Istio to the project.
- Finally, In the sixth phase, we will write this report with the final results.

3. CONTEXTO

A fin de que se entienda la totalidad del proyecto, es menester explicar cada plataforma y arquitectura a la que se hace referencia o uso. Siguiendo un orden cronológico, comenzamos esta sección con la arquitectura monolítica.

3.1. ARQUITECTURA MONOLÍTICA

La arquitectura monolítica integra la totalidad de su funcionalidad bajo un mismo desplegable, es decir, que contiene la lógica del programa y el acceso a los datos, en resumen, toda la capa de usuario bajo la misma plataforma.

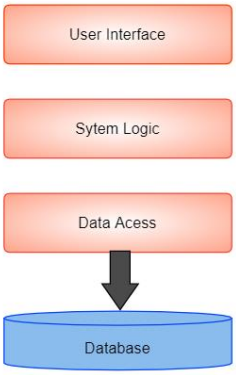


Figura 1- Arquitectura Monolítica

Toda la conexión entre los módulos de la aplicación está corriendo en la misma máquina virtual. Como podemos intuir esto hace que el mantenimiento y la actualización de esta sea una tarea ardua, puesto que todas las conexiones pasan por una única memoria, y esto conlleva que cualquier problema o cambio ocasione la caída y posterior despliegue de la toda la aplicación.

3.2. ARQUITECTURA BASADA EN MICROSERVICIOS

Esta arquitectura, también conocida por sus siglas en inglés MSA (*Micro Services Arquitecture*), es una forma de desarrollo software que se basa en la implementación independiente de cada uno de los servicios de la aplicación.

Esto nos permite, mediante el enfoque de diseño guiado por el dominio (*Domain-Driver Design)*, particionar los servicios con el objetivo de que el desarrollo, comunicaciones, despliegue y producción puedan realizarse sin la dependencia que existe en la arquitectura monolítica.

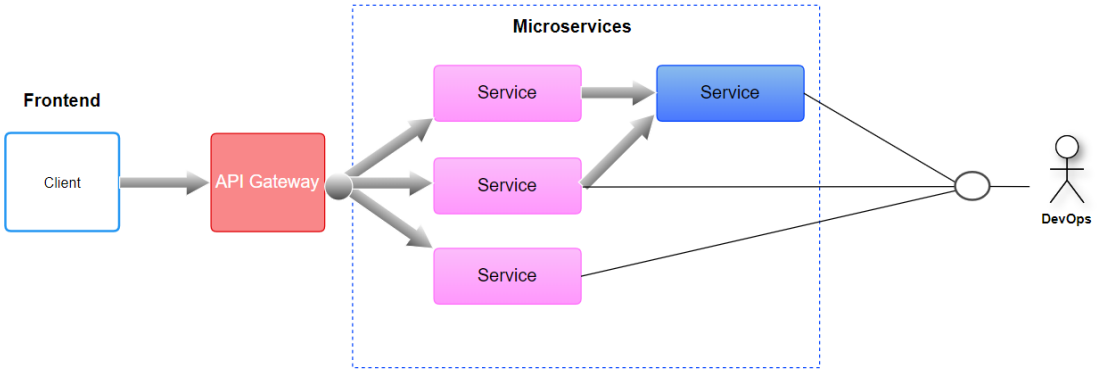


Figura 2- Arquitectura de Microservicios

Como podemos observar en la figura, hay dos tipos de conexiones en esta arquitectura, la primera conecta la interfaz (frontend) con los microservicios a través de una puerta de enlace (Gateway) de API (Application Programming Interface), y la segunda es cualquier conexión entre los servicios que poseen la lógica de negocio, por ejemplo, la que relaciona una base de datos con una aplicación Java.

La arquitectura de microservicios es una aplicación de la programación distribuida, que permite desarrollar sistemas abiertos, escalables, transparentes y tolerantes al fallo, consiguiendo así una producción de forma más continua.

Otra característica de esta arquitectura es que nos permite desarrollar nuestra aplicación en distintos lenguajes de programación, ya que nuestros servicios van a estar aislados y se van a comunicar con mecanismos globales. Como podemos deducir, esto hace que los servicios sean más especializados, es decir, que, al poder elegir distintos tipos de lenguajes, vamos a poder resolver problemas más específicos de una forma más eficiente.

Por último, cabe destacar que en la mayoría de los proyectos el ciclo de vida de las aplicaciones se ve alargado, debido a que como expresamos anteriormente el mantenimiento es más rápido y duradero.

3.3. API REST

Para comprender los conceptos de la arquitectura implementada es importante describir que es una API REST debido a que los microservicios Usuarios, Empleados y Python son de este tipo.

El termino API (*Application Programming Interfaces*) es una cualidad que permite describir la manera en que los programas intercambian información. Por lo tanto, especifican una serie de reglas y detalles para que se puedan comunicarse entre sí. Gracias a esto la trasferencia de datos es de manera estándar.

El termino REST (*REpresentational State Transfer*) tiene su origen en los años 2000 que fue definido por Roy Fielding, el cual es considerado el fundador del protocolo HTTP (*Hyper Text Transfer Protocol*). Este tipo de arquitectura está normalmente orientada al desarrollo web debido a que utiliza el estándar HTTP, usando formatos de mensaje en JSON.

Para que un sistema se considere RESTful debe cumplir los siguientes requisitos:

- **Protocolo Cliente-Servidor:** Este protocolo permite mantener al cliente y servidor poco acoplados. Es decir, el cliente no debe conocer todos los detalles de cómo esta implementado el servidor, y el servidor no se preocupa de como el cliente usa los datos. Por tanto, ni el servidor ni el cliente tienen que recordar algún estado previo. Aunque en algunos casos existen excepciones como aquellas aplicaciones que tienen cache para que el cliente pueda ejecutar la misma petición con respuestas idénticas.
- **Interfaz Uniforme:** Se debe definir una interfaz genérica que permite administrar las peticiones que son producidas por el cliente y servidor. Es decir, cada recurso del servicio REST debe tener una única URI, donde se aplican acciones concretas como GET, POST, PUT, DELETE.
- **Uso de Hipermedios:** Permite explicar la capacidad que tiene una interfaz de aplicaciones, para poder facilitar a los usuarios y clientes las acciones necesarias sobre los datos.
- **Sin estado:** En este caso cada petición que recibe el servidor debe ser independiente. Por lo tanto, no debe mantener sesiones.
- **Cacheable:** Se debe permitir tener un sistema que almacena la cache para evitar tener que pedir varias veces un mismo recurso.
- **Sistema de capas:** Es una arquitectura jerárquica entre los distintos componentes. Estas capas ayudan a mejorar el rendimiento, seguridad y escalabilidad.

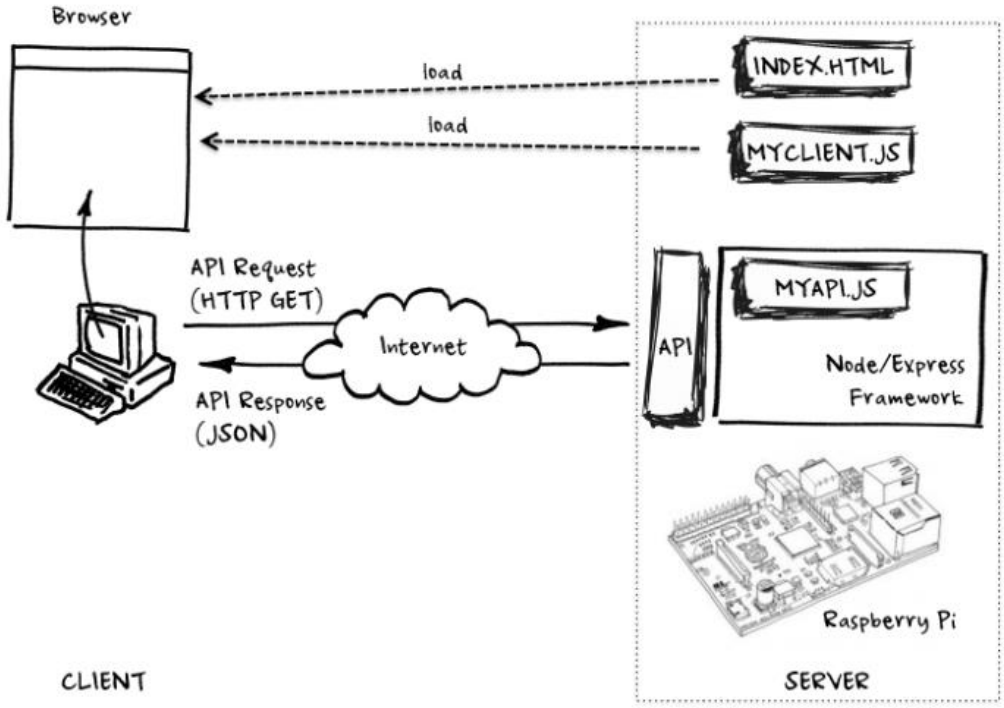


Figura 3- Ejemplo API REST
Fuente: http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2/

Las ventajas de usar una arquitectura REST son las siguientes:

- **Visibilidad, fiabilidad, escalabilidad:** Debido a que el cliente y el servidor no están acoplados permite escalar el producto sin tener demasiadas complicaciones para el equipo de desarrollo. Además, se puede modificar la base de datos siempre que los datos de las peticiones se envíen de forma correcta. Esto permite tener el *front* en un servidor, y el *back* en otro.
- **Separación entre el cliente y el servidor:** Permite migrar el producto a otros servidores, y desarrollar distintos componentes de forma independiente. Por lo que al estar débilmente acoplado cualquier cambio en el servidor se puede realizar de forma fácil.
- **Independencia del tipo de lenguajes o plataformas:** La API REST permite desarrollar servicios en distintos lenguajes como PHP, Python, Java y comunicarse entre sí mediante REST. Por lo tanto, lo hace independiente del tipo de plataforma o lenguaje usado. Lo único que se debe tener en cuenta es el tipo de lenguaje utilizado en el intercambio de información, para que las aplicaciones se puedan entender entre ellas.

El formato de los mensajes en la arquitectura REST es JSON, el cual es un formato de texto que es muy fácil de usar para el intercambio de información. Los archivos JSON contienen solo texto. La sintaxis se centra en pares clave-valor.

Usar este tipo de estructura permite facilitar a los servicios tratar la información, ya que se sigue un formato estándar.


```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customers": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```


Figura 4- Ejemplo JSON

3.4. DEVOPS

DevOps es una metodología que permite cambiar la forma por la cual se administra y se lleva el ciclo de desarrollo de software a nivel técnico. El equipo de operaciones y desarrollo eliminan el trabajo por partes y se comienza a trabajar en una colaboración de dos vías. Estos equipos, cubren todo el ciclo de vida y desarrollo de software, lo que garantiza procesos más seguros y rápidos, ya que permite tener una entrega del alta calidad y productos más confiables. Para lograr este objetivo, se han introducido herramientas que nos va a permitir automatizar las tareas monótonas.

Figura 5- Que es DevOps
Fuente: https://www.xeridia.com/blog/sabes-realmente-que-es-devops

Al incluir DevOps en el proyecto nos tenemos que referir a la integración continua, que es una práctica que permite automatizar tareas cuando se produce un evento. Entre las acciones y procesos que se pueden automatizar están las siguientes:

- Compilación de componentes
- Pruebas unitarias
- Pruebas de integración
- Calidad de código
- Escalado
- Despliegue de software

Los objetivos de la integración continua son automatizar el desarrollo del proyecto, permitir la integración de forma sencilla con distintos componentes de la aplicación y poder automatizar las pruebas.

Las ventajas que nos aporta son las siguientes:

- Antes de implementar una nueva versión, se asegura de que nuestra aplicación se esté ejecutando correctamente.
- Simplifica la corrección de los errores detectados.
- Manifiesta de manera veloz los conflictos que puedan tener los desarrolladores.

3.5. DOCKER

Docker es un software de código abierto que permite virtualizar a nivel del sistema operativo usando contenedores, lo que permite tener una capa adicional de automatización de múltiples aplicaciones en los distintos sistemas operativos.

Por lo tanto, debemos ver la diferencia entre virtualización y contenedores. Virtualizar es crear mediante un programa una traslación virtual de un software, como puede ser un sistema operativo o cualquier tipo de servidor, entre otros.

En cambio, con los contenedores de Docker no se emulan todas las llamadas al sistema operativo y esto conlleva a tener un mejor rendimiento, ya que aprovecha el kernel de la maquina anfitriona.

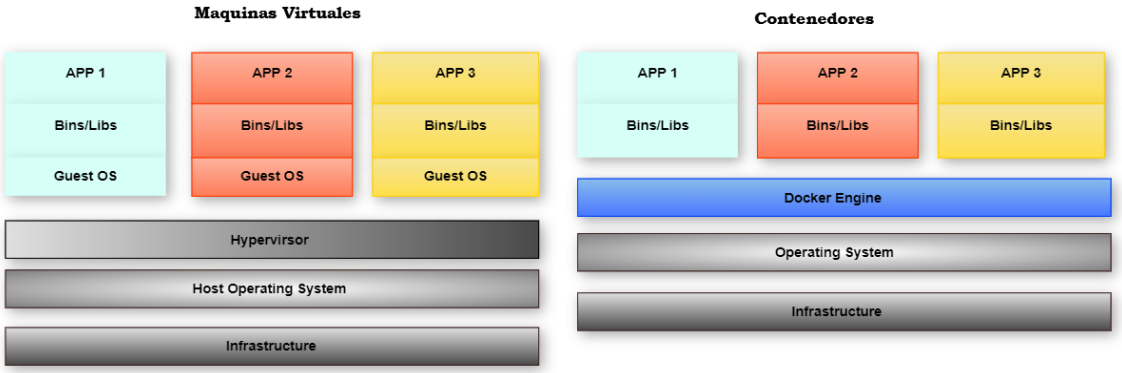


Figura 6.-Comparativa máquinas virtuales y contenedores

Diferencias entre las máquinas virtuales y los contenedores de Docker. Como se aprecia en la imagen la diferencia es que Docker funciona con el sistema operativo del ordenador que lo ejecuta.

Las ventajas de usar Docker son las siguientes:

- Estandarización: Permite ejecutar el mismo software en distintos ordenadores.
- Portabilidad: Ya que los contenedores son portables, nos podemos permitir usarlos en cualquier ordenador sin necesidad de instalar nada adicional.
- Eficiencia: Al usar los contenedores y no realizar todas las llamadas al sistema operativo, utilizando los recursos mínimos, se obtiene una eficiencia del 80% con respecto a la virtualización.
- Despliegue rápido: Docker permite minimizar el despliegue en pocos segundos. Es posible debido a que ejecuta contenedores y no inicia el sistema operativo.
- Seguridad: Docker nos ayuda a que los contenedores que se están ejecutando están aislados entre sí, lo que permite tener control sobre la administración y flujo del tráfico.

3.5.1. COMPONENTES BÁSICOS

IMAGEN

Plantilla donde se incluyen la aplicación y sus distintas librerías, que son usadas para crear los contenedores.

DOCKERFILE

Es un archivo donde se incluye la configuración que permite crear las imágenes. En este archivo se indica el sistema operativo a ejecutar y los distintos comandos que permiten instalar las herramientas necesarias.

```
FROM ubuntu:16.04

RUN apt-get update -y && \ apt-get install -y python3-
pip python3-dev && \ pip3 install --upgrade pip

WORKDIR /app

RUN pip3 install -r requirements.txt

COPY . /app

ENTRYPOINT [ "python3" ]

CMD [ "src/app.py" ]
```

En este DockerFile se indica que se requiere la imagen del sistema operativo Ubuntu:16.04 y sobre esta se ejecutan los comandos para instalar Python.

CONTENEDOR

Son instancias que se producen al ejecutar una imagen donde está la aplicación que hemos desarrollado.

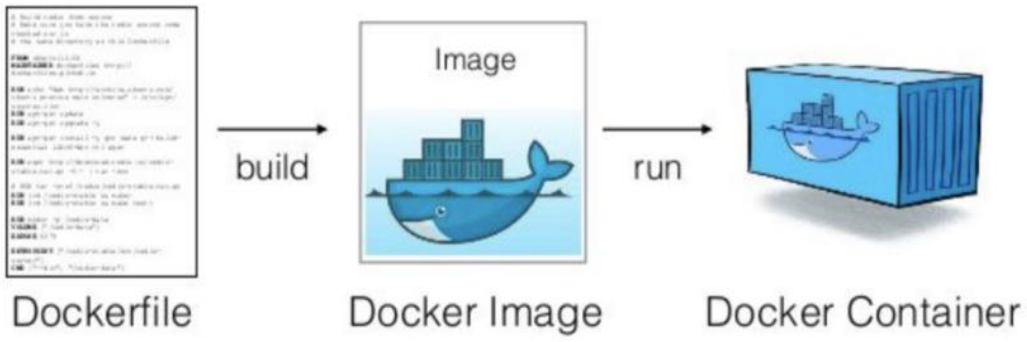


Figura 7- Contenedor
Fuente: https://medium.com/platformer-blog/practical-guide-on-writing-a-dockerfile-for-your-application-89376f88b3b5

VOLÚMENES

Son usados para guardar la información de los contenedores, con el objetivo de que los datos sean persistentes. Ya que al borrar un contenedor se borran sus datos.

LINKS

Son usados para asociar contenedores entre sí que estén dentro del mismo ordenador y se puedan comunicar sin necesidad de exponer el dispositivo que contiene el contenedor.

3.5.2. ARQUITECTURA DOCKER

Docker utiliza una arquitectura cliente-servidor. Donde el cliente de Docker realiza llamadas al demonio de Docker. Se comunican mediante una API REST, a través de los sockets de Unix.

Los componentes básicos son los siguientes:

- ***Docker Client:*** Es la forma principal por la cual los usuarios interactúan con la API REST de Docker, lo que permite gestionar nuestros contenedores, por ejemplo, el comando run.
- ***Docker Daemon:*** Es un demonio que escucha las peticiones de la API para la gestión de los contenedores e imágenes. Un demonio puede comunicarse con otros demonios para poder administrar los distintos servidores de Docker.
- ***Docker Registry:*** Es un componente de Docker donde se almacenan las imágenes que son generadas. Lo que permite que estén disponibles para utilizarlas desde cualquier ordenador.

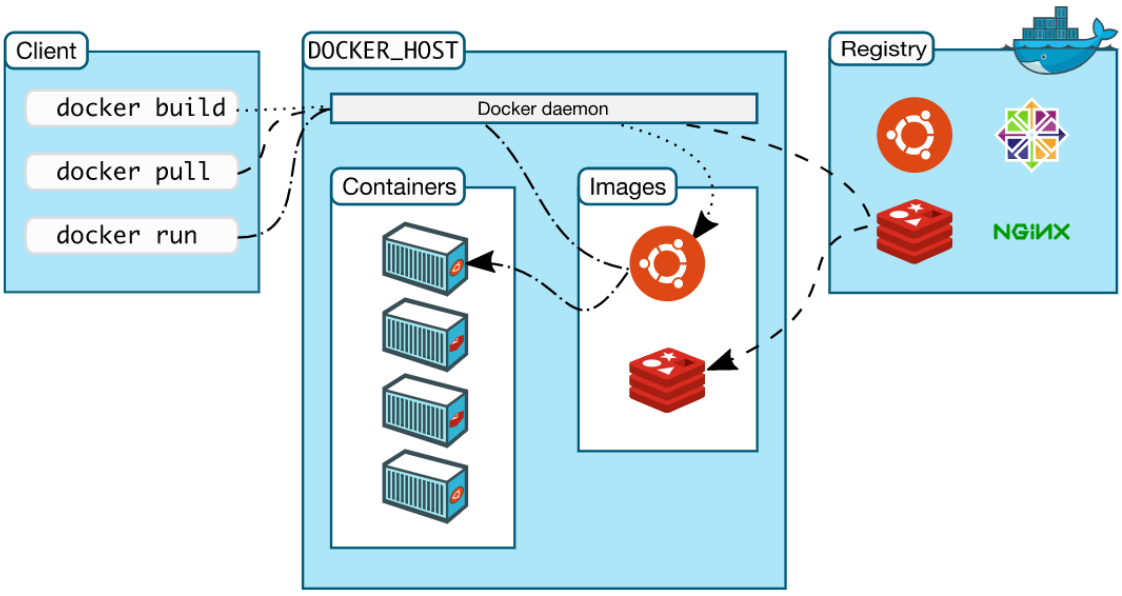


Figura 8- Arquitectura Docker 2
Fuente: https://docs.docker.com/engine/docker-overview/

3.6. ORQUESTADOR DE CONTENEDORES

Los orquestadores nacen de la necesidad de controlar aplicaciones con microservicios, ya que no existe forma manual de controlar una gran cantidad de contenedores a nivel de producción.

Son herramientas de abstracción que nos permiten desplegar un gran número de contenedores dentro de un clúster, estos son espacios administrados por los propios orquestadores y que normalmente están constituidos y controlados por dos tipos de nodos:

- ***Master***: Forman la capa de control y en principio no ejecutan contenedores de usuario.
- ***Worker***: Ejecutan los contenedores que se quieran desplegar dentro del clúster.

La misión principal de un orquestador es seleccionar un servidor de entre un conjunto de máquinas donde ejecutar un contendor. A parte de esta función principal, también pueden llevar a cabo:

- Balanceo de carga: A través de los varios puntos de entrada a los conjuntos de contenedores, se encargan de repartir la carga de las peticiones sobre éstos.
- Escalabilidad: Son capaces de controlar el número de instancias a los contenedores y el número de nodos de un clúster.
- Monitoreo: Esta es una de las funcionalidades más usadas, ya que monitorizar el comportamiento de un sistema conlleva una gran carga de trabajo y coste. Los orquestadores son capaces de observar los contendores controlando posibles fallos y paliándolos.
- Configuración de red: Nos permiten configurar varios parámetros de red, pero ya que vamos a utilizar una malla de servicios, esta funcionalidad no la llegaremos a poner en práctica.

Estas son las funcionalidades más usadas de los orquestadores, pero se les atribuye una lista bastante más larga.

En la actualidad, existen varias implementaciones de orquestadores, por eso a continuación vamos a hacer una comparativa rápida de los tres más conocidos: Docker Swarm, Kubernetes y Apache Mesos.

3.6.1. COMPARATIVA ORQUESTADORES

Comenzamos esta comparativa con el que a nuestro juicio es el más simple, Docker Swarm, este orquestador está integrado en la API de Docker Engine, por lo que nos presenta mayor facilidad para adecuarnos a él, ya que antes hemos trabajado con Docker. Tiene dos grandes inconvenientes, no posee control de fallos y no permite agregar configuraciones nuevas. Puesto que en este trabajo queremos profundizar en las funcionalidades de los orquestadores, no podemos escoger uno que no posea control de fallos.

Una vez descartado una de las posibilidades, analicemos el siguiente, Kubernetes, este es un orquestador completo a la vez que complejo, posee una arquitectura con un

master node y otro *worker*, como ya anticipábamos en la sección anterior, además abarca todas las funcionalidades expuestas.

La última posibilidad que vamos a analizar es Apache Mesos, está basado en el kernel de Linux y presenta múltiples *frameworks* sobre los que se ejecutan los contenedores, es tan completo como Kubernetes, pero dado la creciente popularidad de este último, es por el que hemos optado.

3.6.2. KUBERNETES

Kubernetes es una plataforma de código abierto (*open source*) que nos ayuda a administrar y gestionar nuestros servicios, nace de la experiencia de Google controlando aplicaciones en producción a gran escala.

ARQUITECTURA DE KUBERNETES

Como ya sabemos, Kubernetes divide su arquitectura en dos nodos, aunque cuando corremos en local sólo suele haber un nodo que actúa como *master* y *worker*.

La lógica de la capa de control se fundamenta en *etcd*, una base de datos distribuida que nos asegura la consistencia de los datos usando el algoritmo de consenso Raft. Las peticiones a esta base de datos pasan por la capa REST, que puede ser invocada o accedida a través de los comandos *kubectl*. También posee un módulo de autorización que controla la comunicación entre los comandos y la capa REST. El resto de la lógica la llevan a cabo funciones del *master node* que se ejecutan en bucle y comprueban el estado del clúster.

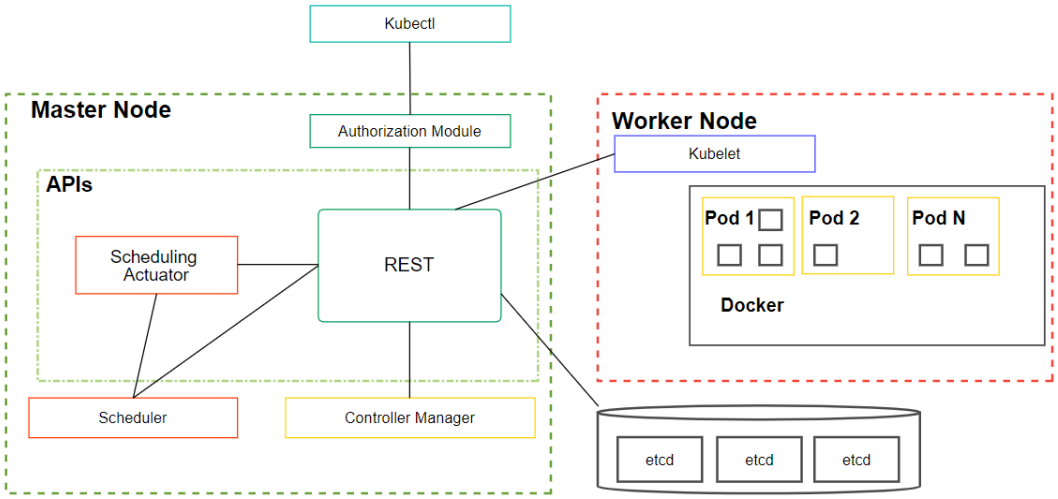


Figura 9- Arquitectura de Kubernetes

En referencia a los *worker nodes*, además de albergar los contenedores que el usuario ha desplegado, poseen dos componentes:

- ***Kubelet***: se comunica con la capa REST con el objetivo de verificar el estado de *etcd*, y así conocer que contenedores tienen asignados. Gracias a esta comunicación son capaces de crear o eliminar contenedores.

- ***Kube-proxy***: se encarga de gestionar la configuración y el estado de la red, creando reglas *iptables*.

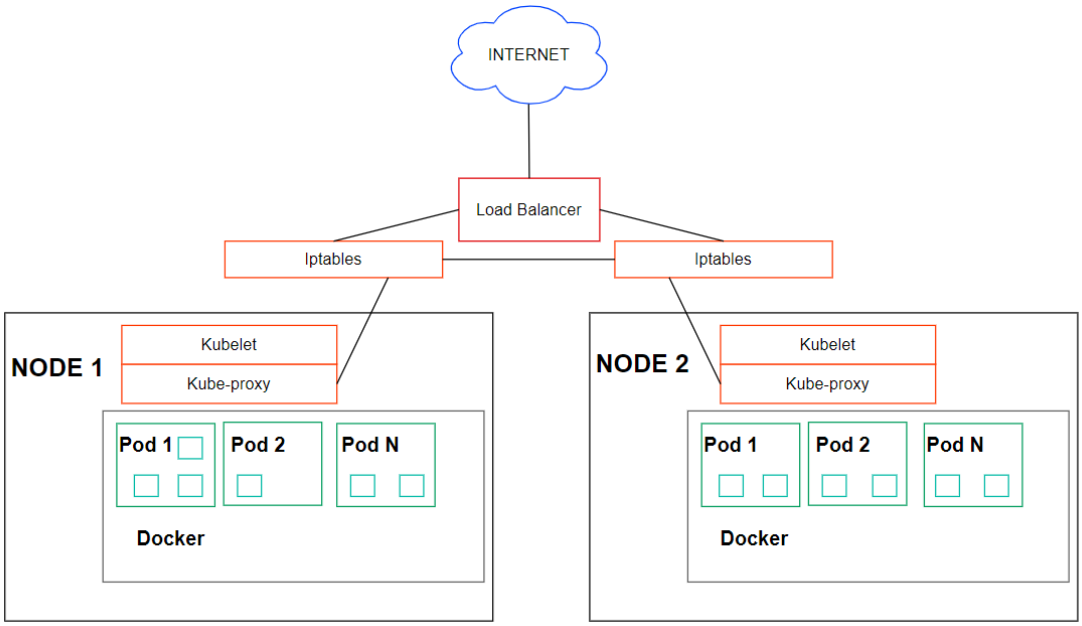


Figura 10- Arquitectura de Kubernetes con Kube-proxy

OBJETOS PRINCIPALES

Son entidades persistentes que representan el estado del clúster, funcionan como un registro de las acciones que ocurren. Estos objetos suelen describir el estado ideal o deseado, que el desarrollador especifica en los archivos de configuración.

NOMBRES E IDENTIFICADORES

Cada elemento que esté en la API REST de Kubernetes se identifica de forma inequívoca por un nombre y un identificador.

NAMESPACE

El clúster físico de Kubernetes puede particionarse en varios clústeres virtuales, denominados *namespaces*. Este es uno de los objetos que nos ayuda en la seguridad y conexión entre los contenedores.

ETIQUETAS Y SELECTORES

Son pares clave-valor que nos permiten identificar componentes del sistema.

LABELS

Son un mecanismo muy potente de filtrado dentro de la arquitectura de Kubernetes, hay que tener especial cuidado ya que algunas veces se utilizan en la configuración de las conexiones.

COMPONENTES PRINCIPALES

Los componentes posibilitan la configuración idónea para el despliegue y control de la aplicación durante su ciclo de vida. Estos componentes son definidos en los archivos de configuración que trataremos en las secciones posteriores.

PODS

Conjunto de contenedores que se ejecutan compartiendo la misma pila (*stack*) de red, volúmenes y memoria. Los contenedores que se ejecuten en el mismo *pod*, podrán comunicarse entre ellos sin necesidad de activar o desplegar otro recurso.

Pueden pasar por varios estados que permiten definir el estado actual del mismo: *Running, Pending, Failed, Suceeded y Unknown*.

REPLICA CONTROLLER/SET

Mecanismo de abstracción sobre un conjunto de *pods*, que tiene como funcionalidad única mantener activo un número especifico de instancias del *pod*. Escalan arriba o abajo estas instancias y las reemplazan si se detecta que no están funcionando correctamente.

DEPLOYMENT

Abstracción sobre los *Replica Controller*, y componente más usado para el despliegue de los *pods*, ofrece actualizaciones continuas (*rolling updates)* y reversiones *(rollbacks)* ya que mantienen un registro histórico sobre las versiones de los contenedores.

SERVICES

Debido a la continua creación y destrucción de los *pods*, es necesario un mecanismo que nos permita definir un extremo (*endpoint*), con el objetivo de conectar interrumpidamente los contendores que permanecen en *pods* diferentes.

SECRETS

Mecanismo que nos permite definir palabras clave o sensibles, con el objetivo de que no sean reveladas. Suele atribuirse a contraseñas, por ejemplo, la de la base de datos. Normalmente son definidos como variables de entorno.

INGRESS

Posibilita la definición de rutas de entrada desde fuera de nuestro ecosistema, esta funcionalidad también la veremos en Istio, y será donde hagamos uso de ella.

VOLÚMENES

Mecanismo que nos permite guardar información persistente, normalmente es usado para las bases de datos. Existen varios tipos, en este proyecto usaremos dos de ellos. Es un elemento muy útil, ya que sobre todo en desarrollo vamos a tener que crear y destruir las instancias a la base de datos, y de esta manera, no nos tendremos que preocupar por la persistencia de los datos.

ARCHIVOS Y EXTENSIONES

Los archivos que definen la configuración del clúster usan el formato YAML (*YAML Ain't Markup Language*) (con extensión- *.yaml* o *.yml*). Kubernetes se encarga de convertir estos archivos a formato JSON (*JavaScript Object Notation*) cuando realiza las llamadas a la API.

Están divididos en varios campos, vamos a definir cuáles de ellos son esenciales para una correcta implementación:

- *apiVersion*: Versión de la API que estás usando al crear el objeto en cuestión.
- *Kind*: Tipo de objeto o componente que quieres desplegar, por ejemplo, "Pod".
- *Metadata*: Datos que permiten identificar unívocamente al objeto, incluyendo una cadena de texto para el *name*, UID, y opcionalmente el *namespace*.
- *Spec*: describe el estado deseado del objeto.

La siguiente figura sirve de ejemplo de todo lo expuesto en esta sección, y es uno de los desarrollados y utilizados en este proyecto. En este momento no vamos a explicarlo en profundidad, ya que lo haremos en los puntos siguientes junto al resto de archivos.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysqldb
spec:
  selector:
    matchLabels:
     app: mysqldb
  template:
    metadata:
      labels:
        app: mysqldb
    spec:
      volumes:
      - name: mysql-pvc
        persistentVolumeClaim:
         claimName: mysql-pvc
      containers:
      - name: mysqldb
        image: mysql:5.7
        #imagePullPolicy: Always
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: root
        - name: MYSQL_DATABASE
          value: db_personas
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-pvc
          mountPath: /var/lib/mysql/
```

Figura 11- Ejemplo de archivo YAML

3.7. MALLA DE SERVICIOS

En la actualidad, ya no es una incógnita o problema el desarrollar los microservicios, ya que contamos con múltiples lenguajes de programación de alto nivel, tampoco lo es el despliegue, ya que existen herramientas como Docker, Minikube, Kubernetes, Openshift, etc. En estos momentos, el punto que provoca más incertidumbre es como controlar la totalidad de la arquitectura de microservicios.

En un principio, las comunicaciones entre los microservicios se administraban mediante la metodología punto a punto, en la que se añadía la funcionalidad adicional para lograr la conexión.

Aunque alguna de estas funcionalidades ya están añadidas en plataformas como Kubernetes o Openshift, no son suficiente viables como para apostar por ellos en el control total.

En el momento en el que se observa que la mayor parte de las conexiones requieren de una configuración parecida y general, se empiezan a desarrollar las mallas de servicios.

Una malla de servicios es una plataforma que permite controlar y monitorizar las conexiones entre los servicios, posibilita el intercambio y el uso simultáneo de datos. Una vez que la introducimos en el ecosistema de nuestra aplicación nos facilita el desarrollo, la producción y el mantenimiento, además evita tiempos de parada (*downtime*) a medida que la aplicación crece. Todo esto es debido a las siguientes afirmaciones:

- Proporciona descubrimiento de servicios, equilibrio de carga, encriptación, gestión de errores, autenticación y autorización, y soporte de patrones como es *circuit breaker*.
- Posee herramientas que facilitan la trazabilidad del sistema.
- Capacidad de coexistir con Docker y Kubernetes, entre otros.
- Se encarga de inyectar un intermediario (*proxy*) de una manera rápida y fácil en nuestros *pods*.

Conociendo todas estas cualidades de las mallas de servicios, parece indispensable introducir en nuestra arquitectura de microservicios estas funcionalidades. Llegados a este punto solo quedaría elegir cuál de las plataformas, que nos proporcionan este servicio, utilizar.

Dado que en el enunciado de este TFG optaban por la elección de Linkerd e Istio, son los que vamos a comparar en esta sección, pero cabe señalar que existen otros softwares que ofrecen esta funcionalidad, cómo, Athos Service Mesh (Google).

3.7.1. COMPARATIVA LINKERD E ISTIO

Linkerd es un proxy de red, de código abierto, creado por la empresa Buoyant, como plataforma pionera en las mallas de servicios. Su principal misión es unir (hacer "*link*", como su propio nombre indica) los diferentes componentes de un sistema.

Linkerd desempeña todas las funcionalidades principales de una malla de servicios, pero no posee un plano de control (*control plane*), es decir, que el sistema no tiene un punto

de control único y común. Esto ocasiona un gran hándicap a nivel de desarrollo y producción.

Conociendo esta descripción, hemos llegado a la conclusión de que la mejor herramienta a utilizar es Istio, puesto que posee plano de control y las demás características de estas dos plataformas son parecidas: mismos protocolos soportados, inyección de *sidecar* manual y automática, mismas funcionalidades, etc.

3.7.2. ISTIO

Istio es una implementación de malla de servicios, de código abierto, y un complemento perfecto para Kubernetes y Docker. Las funcionalidades de Istio se dividen en cuatro grandes grupos: Conexión, Seguridad, Control y Observabilidad. Con estos grupos es capaz de abarcar todos los requisitos de las mallas de servicios que expusimos anteriormente.

ARQUITECTURA DE ISTIO

Istio se divide lógicamente en:

- El plano de datos: se compone de un conjunto de proxys inteligentes (Envoy) desplegados con el patrón *sidecar*. Estos proxys median y controlan toda la comunicación de red entre microservicios junto con Mixer, un centro de telemetría y política de propósito general.
- El plano de control, o *control plane*: gestiona y configura los servidores proxy para encaminar el tráfico.

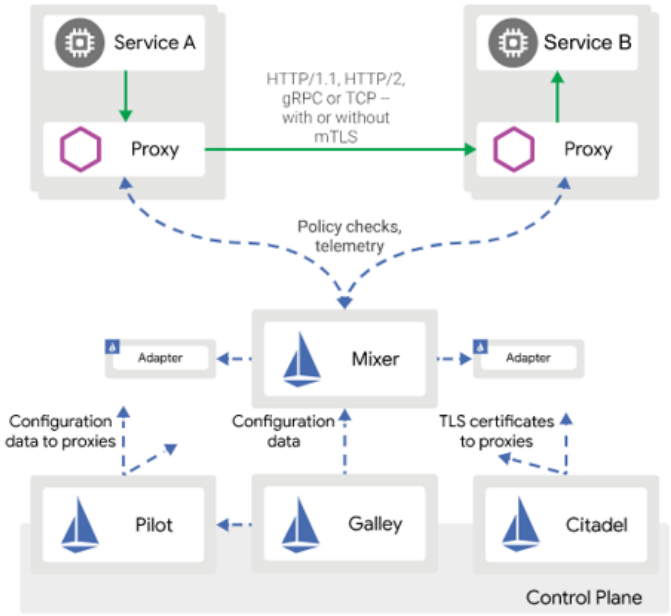


Figura 12- Arquitectura de Istio
Fuente: https://istio.io/docs

PROXY ENVOY

Istio usa una versión extendida del *proxy* Envoy. Es un *proxy* de alto rendimiento desarrollado en C++ para mediar todo el tráfico entrante y saliente de todos los servicios en la malla.

MIXER

Mixer aplica políticas de control de acceso y uso a través de la malla de servicios, y recopila datos de telemetría del *proxy* Envoy y otros servicios. El proxy extrae los atributos de nivel de solicitud y los envía a Mixer para su evaluación.

PILOT

Pilot proporciona descubrimiento de servicio para los *sidecars* Envoy y capacidades de gestión de tráfico para el enrutamiento inteligente. Pilot convierte las reglas de enrutamiento de alto nivel, que controlan el comportamiento del tráfico, en configuraciones específicas de Envoy, y las propaga a los *sidecars* en tiempo de ejecución.

CITADEL

Citadel permite una autenticación sólida de servicio a servicio, y de usuario final, con gestión de identidad y credenciales incorporada. Puede usarse para actualizar el tráfico no cifrado en la malla de servicios.

GALLEY

Galley es el componente de validación, consumo, procesamiento y distribución de configuraciones de Istio. Es responsable de aislar el resto de los componentes de Istio de los detalles, para obtener la configuración del usuario de la plataforma subyacente.

GESTIÓN DEL TRÁFICO

Con el objetivo de administrar el tráfico en la malla, Istio detecta todos los *endpoints* automáticamente en el momento que se integra en el clúster de Kubernetes. Esto se lleva a cabo cuando se conecta al sistema de descubrimiento de servicios, así es capaz de computar su propio registro de servicios.

A través de este registro los Envoy son capaces de dirigir todo el tráfico, enlazando el origen y el destino de las peticiones. Los Envoy ponen en práctica el balanceo de carga, utilizando las múltiples instancias que normalmente tienen los servicios y mecanismos de turnos (*round-robin*).

Istio aporta varias formas de enlazar recursos en diferentes clústeres, *namespaces*, redes, o incluso mallas. Todo esto es posible gracias a los servicios que ofrece su API.

RECURSOS DE LA API DE ISTIO

VIRTUAL SERVICES

Permiten configurar el enrutamiento de las solicitudes a un servicio. Constan de un conjunto de reglas que definen como se deben tratar estas peticiones.

DESTINATION RULES

Son las normas que ayudan a Istio a conocer con exactitud el destino real, sobre el cuál se debe aplicar un *virtual service*. Agrupan todas las instancias de un servicio, adjudicándole un nombre a cada grupo que le defina inequívocamente.

GATEWAYS

Actúan como puerta de enlace sobre la malla, haciendo posible las comunicaciones entre el exterior y el sistema que se encuentra dentro de dicha malla. Se aplican en los servidores Envoy que se van a comunicar con el exterior, normalmente una interfaz o una API REST.

SERVICE ENTRIES

Son las entradas que Istio agrega al registro de servicios, de tal manera que los Envoy conocen cada instancia de dicho registro como un posible destino con el que comunicarse.

SIDECARS

Istio inyecta en cada *pod* un nuevo contenedor con la funcionalidad de un *proxy* Envoy, para administrar todas las peticiones entrantes y salientes de ese *pod*.

TIMEOUTS

Es el tiempo de espera que tienen que respetar los Envoy, para asegurar que hay respuesta a cada petición. Así Istio permite que se reconozcan las respuestas que pueden retrasarse, y finalizar la conexión con el objetivo de que el servidor no se quede indefinidamente esperando dicha respuesta.

RETRIES

Este recurso especifica la cantidad máxima de reintentos que debe llevar a cabo un *proxy* en la conexión de un servicio, si falla en la llamada inicial.

CIRCUIT BREAKER

Se basa en el patrón que acuña se nombre. Es una funcionalidad muy útil, ya que establece el número de veces que una conexión puede fallar antes de que se interrumpa esta. Dicha interrupción puede ser definida de varias maneras, la más común es aquella que se configura de tal manera que pasados unos segundos determinados vuelve a incorporarse la ruta en las conexiones, otra posibilidad es romper la conexión de manera indefinida.

Es muy común que, si un conjunto de instancias recibe muchas peticiones, se produzca un retraso en las respuestas de una de ellas, si es el caso, Istio no permitirá que lleguen más peticiones a esa instancia con el fin de que la sobrecarga finalice.

FAULT INJECTION

Es un mecanismo de prueba que mide la capacidad de recuperación de fallas en la aplicación. Existen dos tipos de fallas:

- Delays: Son retrasos en las peticiones de una conexión.
- Aborts: Fallas de choque como pueden ser el envío de códigos de errores típicos de las peticiones HTTP

SEGURIDAD EN ISTIO

Las características de Istio a nivel de seguridad son una de las razones por la cual esta plataforma se ha hecho tan popular en el sector. Es de conocimiento público la dificultad que supone tratar con certificados, autoridades de certificación (CA), cifrado, autenticación, autorización y auditoría en una aplicación que no tenga integrada una malla de servicios. Istio ofrece todo esto dentro de sus múltiples configuraciones.

La identidad de Istio es el primer paso en su seguridad, se basa en el intercambio de credenciales de las dos partes en una comunicación, con el fin de asegurar la autenticación mutua. Esto es posible ya que los proxys Envoy trabajan en conjunto con *istiod*, para automatizar la rotación de claves y certificados.

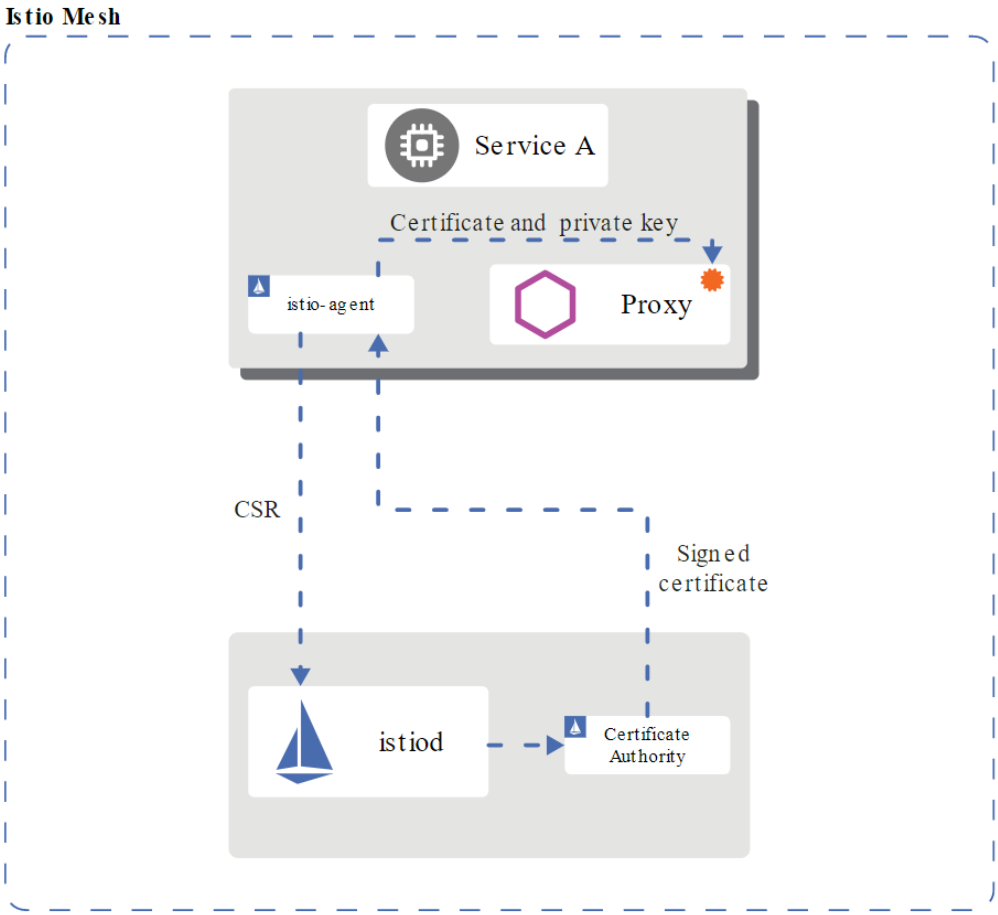


Figura 13- Rotación de certificados Istio
Fuente: https://istio.io/docs

Istio proporciona dos tipos de autenticación:

- *Peer authentication*: Se utiliza para asegurar la autenticación entre pares de servicios. Istio ofrece el TLS mutuo como solución para este tipo.
- *Request authentication*: Se usa para verificar la identidad del usuario final a través de las credenciales adjuntas en la petición.

Además, tenemos la posibilidad de diferenciar la configuración de las autenticaciones de diferentes formas: permisivo, deshabilitado, mutuo y estricto. Ayuda a los servidores a aceptar el tráfico sin formato, mutuo o ambos.

Los proxys de Istio son los que se encargan de la autorización de solicitudes. Controlan un motor que administra peticiones en tiempo real, evaluando el contexto de cada solicitud y devuelve un resultado que puede ser "*allow*" o "*deny*". La siguiente figura, proporcionada por la web oficial de Istio, muestra este proceso de manera más visual.

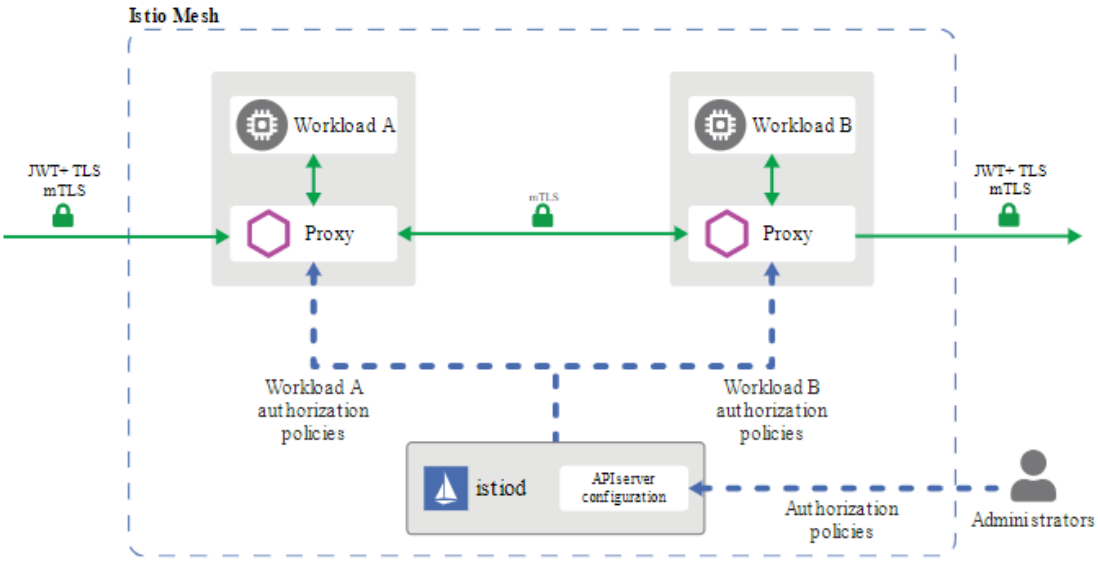


Figura 14- Arquitectura de autorización
Fuente: https://istio.io/docs

OBSERVABILIDAD EN ISTIO

Istio ofrece múltiples herramientas para mejorar la observabilidad, aunque Kiali, Prometheus y Grafana son los ejemplos más utilizados. Istio es capaz de generar un conjunto de métricas de servicio. Además de todo esto, nos ofrece opciones para aplicar la trazabilidad en nuestra malla de servicio, de tal manera que nos permite ver como una petición circula por los *proxies*.

ARCHIVOS EN ISTIO

Para terminar con esta sección, cabe señalar que los archivos para configurar todo lo que hemos expuesto en esta parte, usan el mismo formato (YAML) que los que utilizamos en Kubernetes. Esto hace más llevadero el aprendizaje de esta amplia plataforma, ya que estamos familiarizados con este formato.

4. METODOLOGÍA

Para realizar el proyecto se ha elegido usar una metodología ágil, que permite el desarrollo flexible y evolutivo en toda la duración del proyecto.

La idea es tener *sprints* de dos semanas, que va a permitir realizar pruebas sobre el desarrollo del software. Se involucra al *Stakeholder* para que se pueda comprobar y verificar el desarrollo del proyecto.

4.1. METODOLOGÍA ÁGIL

La metodología elegida es Scrum debido a que el número de personas que realizan este proyecto es de dos y dicha metodología se adapta perfectamente porque, en parte, se pensó para equipos reducidos.

Scrum es una metodología que trabaja con procesos agiles y tiene un ciclo de vida que es incremental e iterativo. Donde se obtienen resultados rápidamente y se puede entregar parte del producto de forma periódica, logrando así encontrar soluciones óptimas.

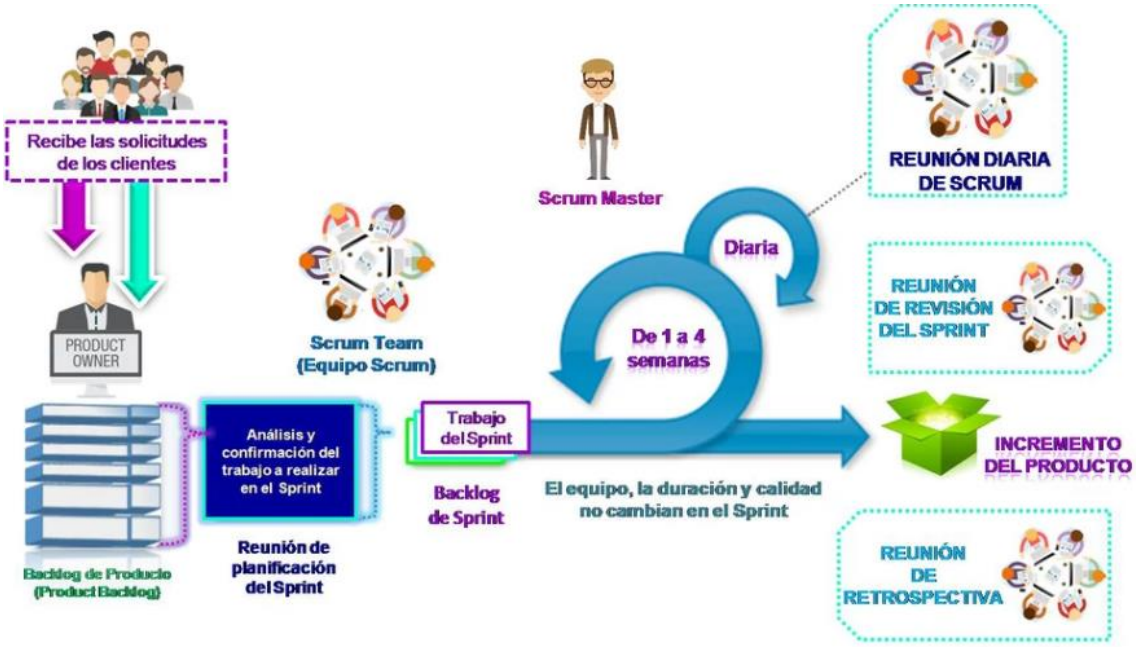


Figura 15- Ejemplo de metoología Scrum

4.2. HERRAMIENTAS USADAS

Las herramientas usadas para seguir dicha metodología son las siguientes:

- **Microsoft Planner:** Es una aplicación con un tablero Kanban que nos permite incluir las tareas a realizar y asignarlas a los respectivos responsables. Nos ofrece ciertas características que nos permiten estar siempre informados.
- **Microsoft Teams:** Es un software basado en un chat de trabajo, que nos permite realizar reuniones. El beneficio de usarlo es que viene integrado con Planner.

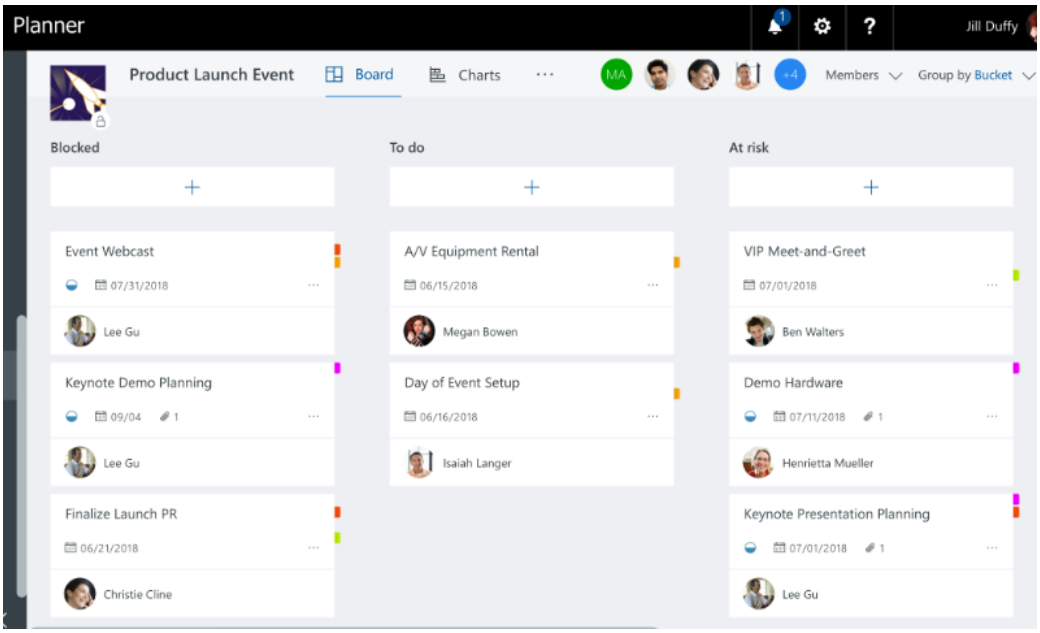


Figura 16- Ejemplo de Microsoft Planner

4.3. ORGANIZACIÓN DEL EQUIPO

La organización del equipo es la basada en la metodología Scrum, teniendo las siguientes reuniones:

- **Reunión de planificación:** Se realiza al inicio de cada sprint, en esta sesión se establece la planificación y objetivos a realizar durante el sprint de dos semanas.
- **Reunión diaria:** Se realiza reuniones de 15 minutos donde se exponen que actividad se realizará en el día y los problemas obtenidos del día anterior. Se establece un plan de trabajo para el próximo día.
- **Reunión de revisión:** Se realizan al final del sprint y se comentan los objetivos que se han completado y los que no.
- **Reunión de retrospectiva:** Se realiza al finalizar el sprint donde se ponen los puntos de vista del equipo, los inconvenientes del sprint y se obtienen las posibles mejoras.

El equipo está conformado por lo siguiente:

- *Stakeholder:* Universidad Complutense de Madrid.
- *Product Owner:* Ricardo Cabrera Lozada y Raúl Diego Navarro.
- *Scrum Team:* Ricardo Cabrera Lozada y Raúl Diego Navarro.

5. DEFINICIÓN DEL ÁMBITO

5.1. LENGUAJES DE PROGRAMACIÓN

Con el fin de demostrar una de las ventajas que poseen las aplicaciones basadas en microservicios, hemos desarrollado una aplicación utilizando múltiples lenguajes de programación, los cuales vamos a exponer en esta sección.

PYTHON

Es un lenguaje de programación interpretado, no compilado, que usa tipado dinámico, fuertemente tipado, multiplataforma y multiparadigma. Podemos encontrarlo corriendo en servidores, en aplicaciones iOS, Android, Linux, Windows o Mac.

Es el lenguaje que normalmente se escoge para desarrollar aplicaciones de Big Data e Inteligencia Artificial (IA), y dado que en la actualidad estas prácticas están en auge hemos querido agregar a nuestro proyecto este lenguaje tan versátil.

En particular, hemos usado su variante Flask, que es un *framework* minimalista que permite crear aplicaciones web de una forma rápida e intuitiva.

Figura 17- Python

Figura 18- Python Flask

JAVA - SPRING BOOT

Spring Boot es una tecnología del grupo Spring que nos facilita el crear un proyecto en Maven, descargar las dependencias necesarias, programar nuestra aplicación en Java y desplegar nuestro servidor.

Es uno de los *frameworks* que actualmente se están usando más para desarrollar API REST, esto se debe a que Spring Boot posee un módulo que autoconfigura todos los aspectos de nuestra aplicación, para poder simplificar a la mínima la definición de esta.

Spring Boot nos proporciona una interfaz para la creación de aplicaciones, donde podemos seleccionar las herramientas que vamos a usar en el proyecto y el propio Spring Boot crea todas las dependencias.

Figura 19- Spring Boot

PHP

Lenguaje de programación originalmente diseñado para el preprocesado de texto plano en UTF-8, pero actualmente en la mayoría de los casos se emplea en el desarrollo web de contenido dinámico. En este proyecto se va a utilizar para la implementación del servicio que contiene la interfaz web.

Figura 20- PHP

SQL

Es un lenguaje de dominio específico, diseñado para administrar, y recuperar información de sistemas de gestión de bases de datos relacionales. Una de sus principales características es el manejo del álgebra y cálculo relacionales.

Figura 21- SQL

5.2. HERRAMIENTAS SOFTWARE

SUBLIME TEXT

Versión: 3.2.2.

Definición: Es un editor de texto y código fuente escrito en C++ y Python.

Motivo: Hemos usado la versión gratuita de esta aplicación para implementar los archivos YAML.

NOTEPAD ++

Versión: 7.8.5.

Definición: Es un editor de texto y código fuente, de código abierto, con soporte para múltiples lenguajes

Motivo: Hemos utilizado este software para editar consultas SQL, elaborar listas de comandos en Docker, Kubernetes e Istio dada la amplia longitud y variedad de estos.

MySQL WORKBENCH

Versión: 8.0.19.

Definición: Es una herramienta visual de diseño de bases de datos en SQL.

Motivo: Nos ha permitido implementar de manera más fácil nuestra base de datos, incluyendo las relaciones entre las tablas y la inyección de datos.

POSTMAN

Versión: 7.24.0

Definición: Es una herramienta diseñada para las pruebas de las API REST, dado que estas carecen de interfaz gráfico.

Motivo: Hemos utilizado esta herramienta para probar las API REST que hemos implementado en los diferentes lenguajes.

INTELLIJ IDEA

Versión: 11.0.4.

Definición: Es un entorno de desarrollo desarrollado por la compañía JetBrains.

Motivo: Utilizado para la implementación de los servicios elaborados con Spring Boot.

JETBRAINS PHPSTORM

Versión: 2020.1.1

Definición: Es un entorno de programación en PHP desarrollado por la compañía JetBrains.

Motivo: Es el entorno donde hemos desarrollado la interfaz web.

JETBRAINS PyCHARM

Versión: 2020.1.1

Definición: Es un entorno de programación en Python desarrollado por la compañía JetBrains.

Motivo: Es el entorno donde hemos desarrollado los dos servicios que tenemos en Python.

VISUAL CODE

Versión: 1.41.1

Definición: Es un editor de código fuente desarrollado por Microsoft para Windows, Linux y Mac.

Motivo: Hemos utilizado este software para la implementación de los servicios, y como gestor de los contenedores en Docker, ya que posee varios *plugins* que facilitan el desarrollo de estos.

DOCKER DESKTOP

Versión: 2.2.0.5

Definición: Es una herramienta que ofrece una forma de configurar Docker y Kubernetes de forma rápida.

Motivo: Hemos utilizado este software para configurar los recursos adecuados para ejecutar Docker y Kubernetes en local.

LIBREOFFICE

Versión: 6.3.1.2.

Definición: Es un paquete de software de oficina libre y de código abierto desarrollado por The Document Foundation.

Motivo: Hemos hecho uso de su procesador de texto para la elaboración de esta memoria.

MICROSOFT OFFICE

Versión: 17.0.

Definición: Es una suite ofimática desarrollada por Microsoft.

Motivo: Hemos hecho uso de su paquete de Word, para editar esta memoria.

XAMPP

Versión: 7.4.3.

Definición: Es un paquete de software libre, que consiste principalmente en el sistema de gestión de bases de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de *script* PHP y Perl.

Motivo: Hemos utilizado esta herramienta principalmente en la base de nuestra implementación para probar la correcta implementación de nuestros servicios, antes de comenzar a usar Docker.

GOOGLE DRIVE

Versión: Online.

Definición: Es un servicio de alojamiento de archivos que fue introducido por la empresa estadounidense Google el 24 de abril de 2012. Es el reemplazo de Google Docs.

Motivo: Hemos utilizado este software para el control de versiones en nuestra implementación de servicios y desarrollo de memoria, además de como copia de seguridad.

GITHUB

Versión: Online.

Definición: Es una plataforma de desarrollo colaborativo de software.

Motivo: Lo hemos utilizado como herramienta para la actualización simultánea de cambios en nuestro proyecto.

DISCORD

Versión: 21.3.

Definición: Es una aplicación freeware de VoIP. Diseñada para la comunicación por audio y video.

Motivo: Dada las circunstancias extraordinarias de este año, hemos utilizado este software para las reuniones online del equipo.

CACOO

Versión: Online.

Definición: Editor de gráficos online, herramienta en auge para diseño de diagramas y demás elementos gráficos

Motivo: Hemos usado este software para diseñar la mayor parte de las figuras y diagramas de esta memoria.

6. PRESUPUESTO Y RECURSOS

En esta sección vamos a tratar todos los tipos de recursos que hemos consumido y necesitado en el diseño e implementación del proyecto. Debido a que el equipo está formado por solo dos integrantes particulares, y no hemos tenido relación con ninguna empresa, estos recursos se ven condicionados, y para extrapolarlo a un proyecto más grande habría que tener esto en cuenta.

6.1. RECURSOS HUMANOS

Como ya hemos anticipado el equipo está formado por dos personas particulares. Además, está supervisado por el tutor de TFG. Aunque no pertenecemos a ninguna empresa, vamos a tratar los datos como si fuera así.

El sueldo medio de un ingeniero informático, desarrollador de aplicaciones, en Madrid, ya que es la ciudad donde hemos estudiado, y de perfil júnior, debido a que no tenemos más de dos años de experiencia en el sector, es de 33.000€ al año.

Como veremos en la sección de planificación, este trabajo se empezó la primera semana de septiembre en 2019, y se va a terminar a finales de junio de 2020. Eso hace un total de 10 meses trabajando en la aplicación. Por tanto, el sueldo medio que cobraríamos sería de 27.500€ aproximadamente.

Una vez que tenemos el sueldo medio aproximado, pasamos a justificar las horas y tareas realizadas en el proyecto, que cubrirían la parte del contrato ficticio que estamos planteando en este punto. En la columna de responsables vamos a discernir entre varios escenarios:

- Si solo hay un “1”, se refiere a que la tarea ha sido realizada solo por un desarrollador, por tanto, el coste sería computado solo a una persona.
- Si hay un “2”, se refiere a las dos personas de este grupo.

Por último, he de añadir que el coste se ha calculado mediante la siguiente fórmula:

Coste = Hora trabajada * 30€.

Siendo 30€ de media lo que cobra un informático con las características anteriores.

TAREA	Horas	Responsables	Coste € Aprox.
Formación	350		
Cursos Online	110	2	3.300
Documentación	240	2	7.200
Diseño, Implementación y Pruebas	105		
Servicios Spring Boot	40	2	1.200
Servicios Python	25	1	750
Servicios PHP	30	1	900
Servicio SQL	10	1	300
Docker	68		
Instalación	3	2	90

Implementación DockerFile	20	2	600
Despliegue	30	2	900
Test y Cambios	15	2	450
Kubernetes	156		
Instalación	6	2	180
Implementación archivos configuración	80	2	2.400
Despliegue	30	2	900
Test y Cambios	40	2	1.200
Google Cloud	43		
Creación	3	2	90
Despliegue	25	2	750
Test	15	2	450
Istio	405		
Instalación	10	2	300
Inyección Sidecar	15	2	450
Despliegue 1	8	2	240
Test y Cambios 1	10	2	300
Análisis	40	2	1.200
Alcance	35	2	1.050
Implementación archivos configuración 1	115	2	3.450
Despliegue 2	10	2	300
Test y Cambios 2	20	2	600
Agregar Funcionalidades	20	2	600
Implementación archivos configuración 2	95	2	2.850
Despliegue 3	12	2	360
Test y Cambios 3	20	2	600
Reuniones	10		
Equipo	--	2	Esta computado en las demás tareas
Documentación	450		
Diseño	5		
Estructura	20		
Elaboración	200		
Corrección	75		
Cambios	50		
Revisión Final	100		
Total	1546		46.380

Tabla 1- Recursos humanos

Como podemos observar el mayor gasto de la implementación se produce en el diseño, desarrollo y gestión de la malla de servicio. Al coste total habría que añadir el precio de

los cursos en los que hemos participado. Estos cursos han sido desarrollados en la plataforma *online* OpenWebinars.

6.2. RECURSOS MATERIALES

Los costes de los recursos materiales son todos aquellos que están relacionados con el puesto de trabajo y el uso de material.

En este escenario tenemos dos recursos principales que son los equipos usados por los dos miembros del equipo.

Producto	Unidades	Especificaciones	Precio
HP Notebook	1	i5 8th, 16GB RAM, 500GB SSD	800€
Lenovo ThinkPad	1	i7 8th, 16GB RAM, 500GB HDD, 500GB SSD	1.200€
Total	2		2.000€

Tabla 2- Recursos materiales.

6.3. RECURSOS SOFTWARE

Los recursos digitales se resumen en los softwares expuestos en la sección anterior. Habría que añadir las licencias que hemos conseguido de forma gratuita, debido a los convenios que posee activamente nuestra facultad, por ejemplo, con la compañía JetBrains.

Todo los demás han sido recursos *open source* que no suponen un gasto económico.

6.4. RECURSOS CLOUD

Google ofrece un período de tiempo gratuito en su plataforma Google Cloud Kubernetes, por un valor de 300$ por cada cuenta, es decir, 600$ ya que somos dos desarrolladores.

6.5. OTROS RECURSOS

Otro recurso esencial e indirecto sería el sitio de trabajo, hasta febrero había sido las instalaciones de la Facultad de Informática de la Universidad Complutense de Madrid. Posteriormente, dado las características del confinamiento ha sido la propia casa de cada uno de los participantes.

Por último, cabe destacar los gastos que no se suelen contabilizar y que también son indirectos. Gasto de electricidad, agua y gas que supondría el desarrollo y producción de un proyecto como el que estamos manejando en este trabajo.

7. PLANIFICACIÓN

7.1. PLANIFICACIÓN INICIAL

Como ya expusimos en la introducción, este proyecto se ha dividido en seis fases claves para su evolución, y dado que hemos decidido que la metodología empleada sea ágil, con *sprints* de dos semanas, vamos a mostrar en esta sección las tareas que hemos realizado y el tiempo que nos ha llevado.

El trabajo comenzó en la primera semana de septiembre de 2019, con la formación necesaria, se ralentizó a finales de diciembre y principios de enero, debido a los exámenes del primer cuatrimestre que tuvimos que realizar los dos participantes del equipo. Y va a terminar a finales de junio con la elaboración de esta memoria y presentación de esta.

Antes de exponer un resumen de los *sprints* realizados en el proyecto, cabe señalar que la planificación que vamos a exponer es aquella a la que nos hemos tenido que adaptar dada la situación excepcional de este curso, puesto que, al inicio de este estudio la establecimos de diferente manera.

7.2. SPRINTS Y FECHAS CLAVE

PRIMEROS DOS SPRINTS (2 de septiembre – 29 de septiembre)

En este período de tiempo, comenzamos la formación online en la web OpenWebinars. Nos centramos en los cursos dedicados a la implementación de nuestros servicios:

- Python 3 desde cero.
- Flask Mini-Framework Python.
- Arquitecturas monolíticas y microservicios.
- Spring Boot y Spring MVC.
- PHP.

Además, leímos gran cantidad de artículos de arquitectura de microservicios, y tecnologías Spring, con el fin de tener los conocimientos suficientes para diseñar y desarrollar los servicios, y sus respectivas configuraciones.

TERCER Y CUARTO SPRINT (30 de septiembre – 3 de noviembre)

Pasamos a la formación de cara a la arquitectura de Docker, Kubernetes e Istio, continuando con la formación online en la misma plataforma, con cursos como:

- Curso de Istio.
- Introducción a Docker.
- Docker para desarrolladores.
- Kubernetes para desarrolladores.

Durante estos *sprints*, el objetivo principal era entender y conocer la mayor parte de la información de las páginas oficiales y blogs más importantes en torno a Docker, Kubernetes e Istio.

QUINTO SPRINT (4 de noviembre – 17 de noviembre)

En este *sprint* nos centramos en aplicar los conocimientos apropiados para diseñar la aplicación, teniendo ya en cuenta las funcionalidades que iba a tener nuestra malla de servicios.

Una vez que finalizamos el diseño de la aplicación, comenzamos a desarrollar los servicios acordados.

SEXTO Y SÉPTIMO SPRINT (18 de noviembre – 15 de diciembre)

Durante las fechas indicadas finalizamos el desarrollo de los servicios que no habían sido implementados en su totalidad. Además, comenzamos con la instalación de Docker, desarrollo de los Dockerfiles de cada uno de los servicios y despliegue de la aplicación.

OCTAVO, NOVENO Y DÉCIMO SPRINT (16 de diciembre – 19 de enero)

En este período se ralentizó la evolución del proyecto debido a los exámenes como anteriormente explicamos. Realizamos las pruebas correspondientes de Docker y finalizamos esta fase con los cambios pertinentes en la implementación de los servicios y de los Dockerfiles.

UNDÉCIMO Y DUODÉCIMO SPRINT (20 de enero – 16 de febrero)

Comenzamos con la instalación de Kubernetes, probando diferentes configuraciones, como Minikube, o la opción que viene por defecto en Docker.

Una vez que acordamos una configuración igual, para los dos participantes, seguimos con la implementación de los recursos de Kubernetes: *Deployments, Services, Secrets, Volumes,* etc.

Desplegamos la aplicación en Kubernetes, además, terminamos este período con el comienzo de la integración en Google Cloud.

DÉCIMOTERCER SPRINT (17 de febrero – 1 de marzo)

Este *sprint* fue clave para la evolución del trabajo, ya que terminamos la fase de Kubernetes y Google Cloud, dando por finalizada los cambios en esta plataforma. Además, empezamos con la instalación y primeros ajustes de Istio.

14º - 18º SPRINT (2 de marzo – 26 de abril)

Este es el período más largo de la implementación, contiene:

- La inyección manual del patrón *sidecar* en los *pods*.
- El ajuste automático para este patrón en las versiones 2.0 de dos de nuestros servicios.
- Elaboración de las *DestinationRules* pertinentes.
- Desarrollo de los *VirtualServices*.
- Implementación de las configuraciones TLS de la malla de servicio.
- Tres despliegues clave diferentes.
- Pruebas en la malla de servicio.
- Aumento de las funcionalidades.
- Últimas pruebas en la aplicación.

Además de lo citado, comenzamos el diseño y la estructuración de esta memoria como documentación del proyecto.

ÚLTIMOS SPRINTS (27 de abril – 23 de junio)

Estos *sprints* están íntegramente dedicados a la elaboración de esta memoria como documentación de este trabajo.

7.3. DIAGRAMA DE GANTT

En este apartado vamos a mostrar el diagrama de Gantt que justifica el tiempo de desarrollo de este proyecto.

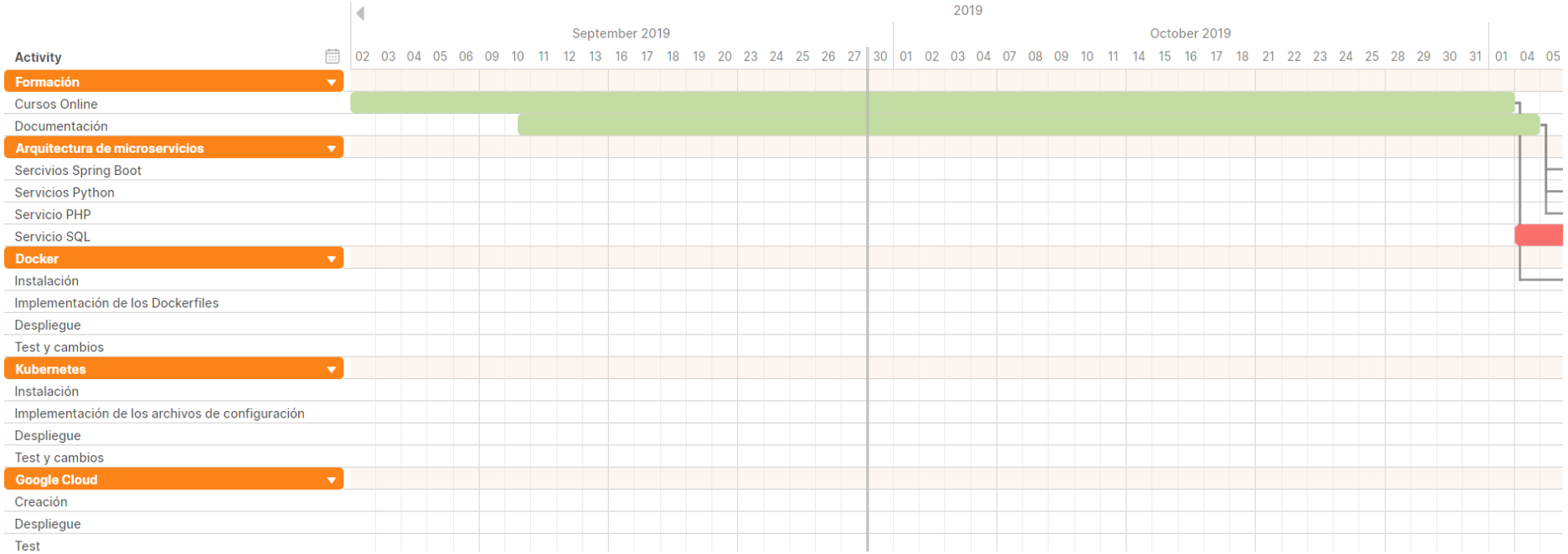


Figura 22- Diagrama de Gantt Parte 1

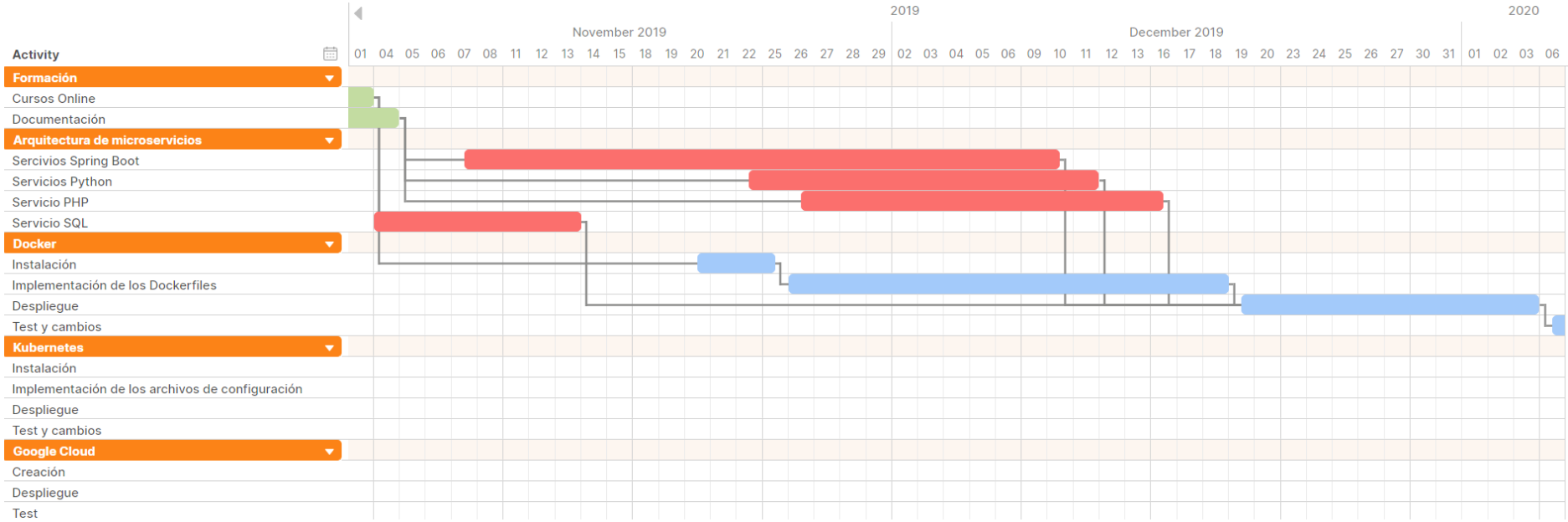


Figura 23- Diagrama de Gantt Parte 2

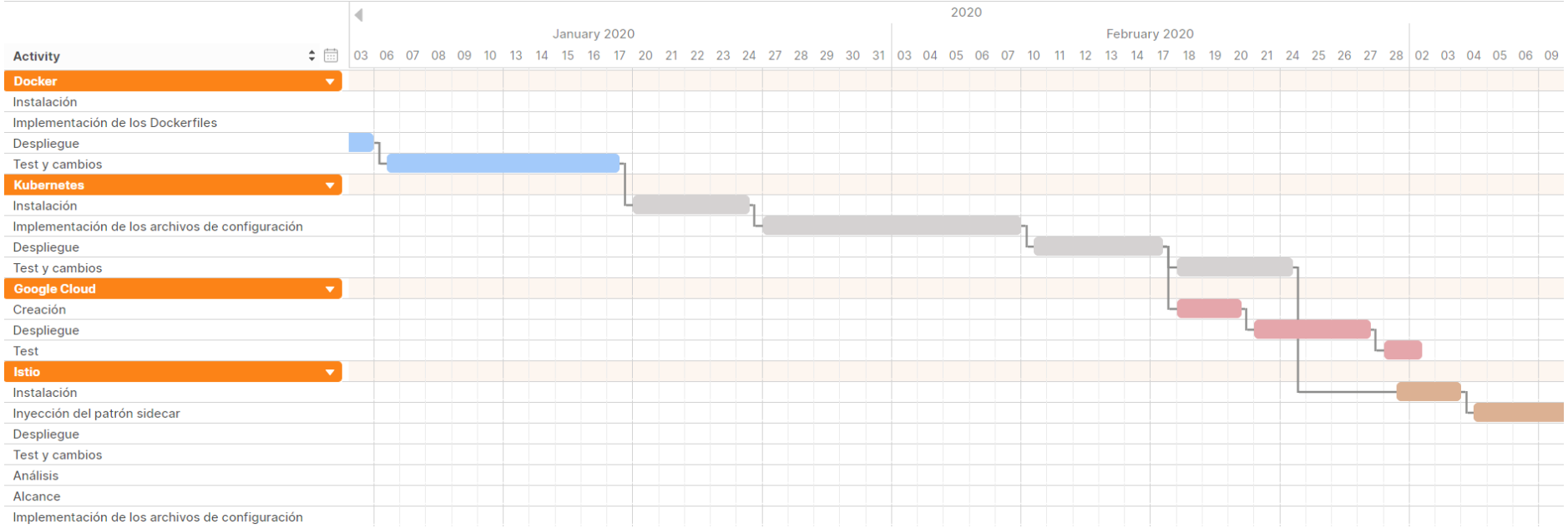


Figura 24- Diagrama de Gantt Parte 3

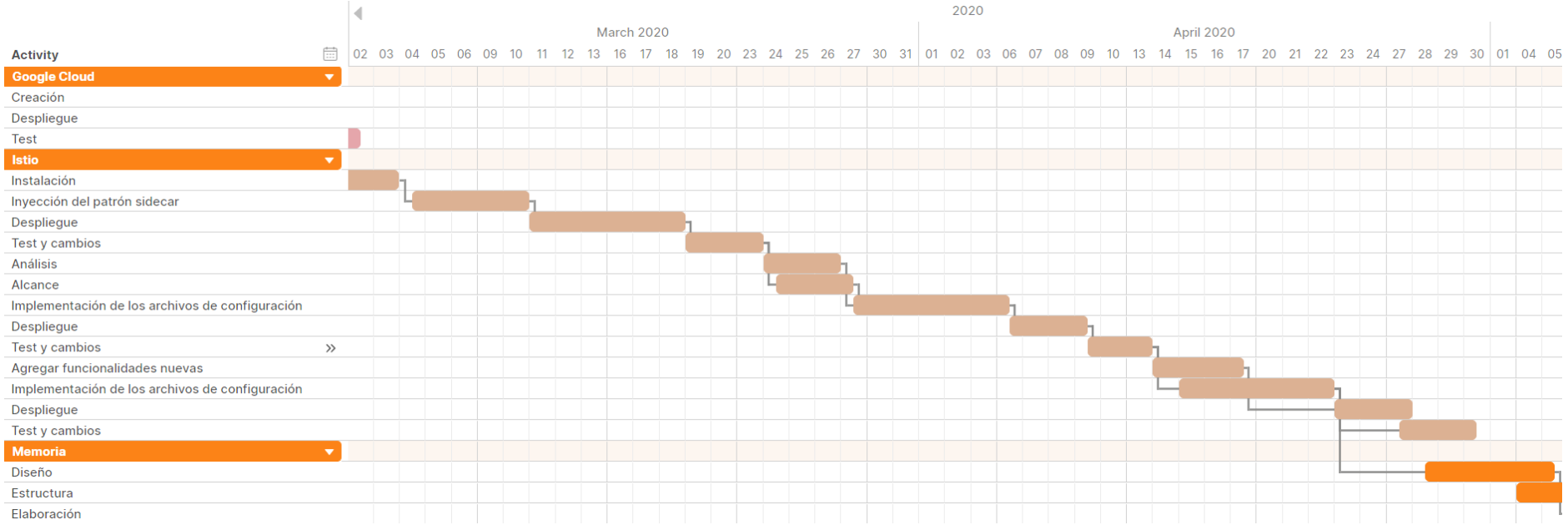


Figura 25- Diagrama de Gantt Parte 4

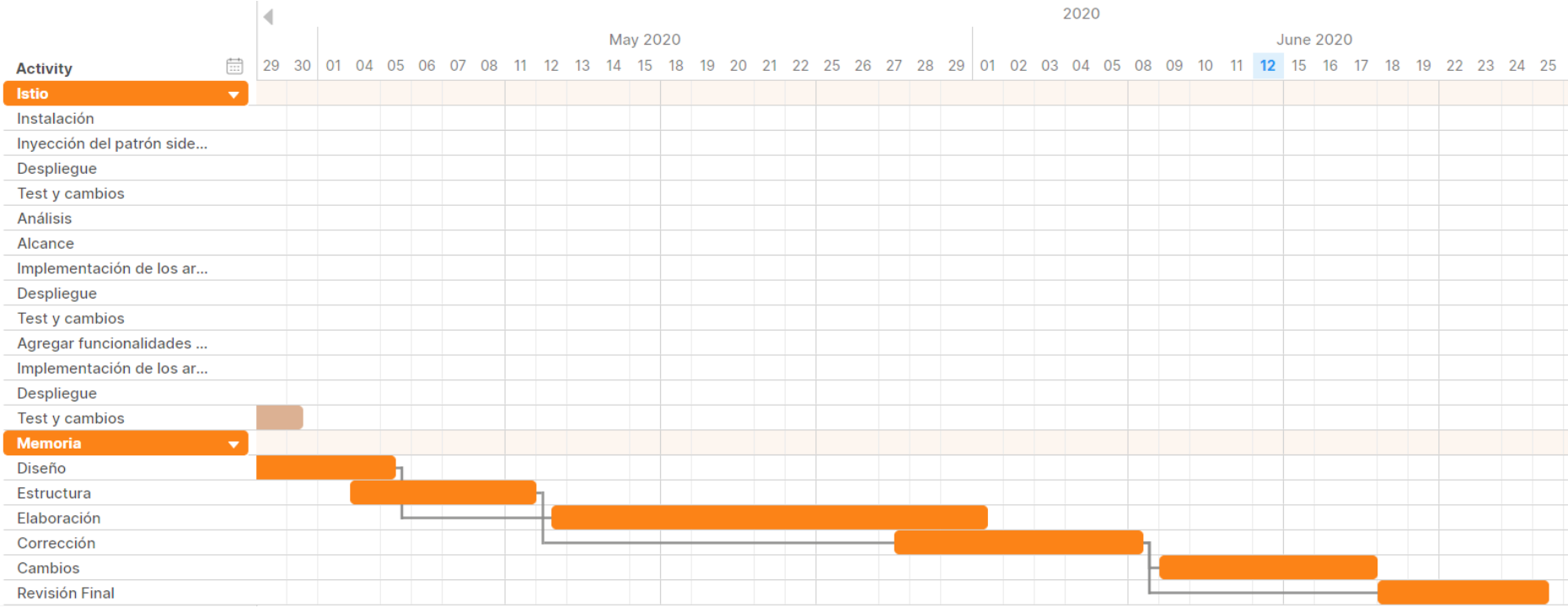


Figura 26- Diagrama de Gantt Parte 5

La leyenda de este diagrama es:

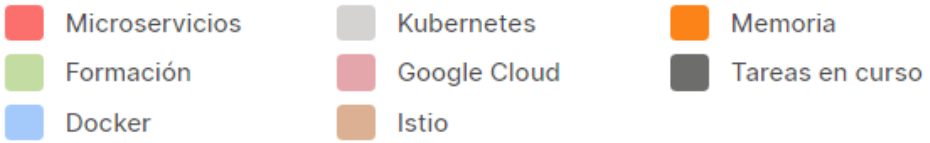


Figura 27- Leyenda de los Gantt

8. DISEÑO DE LA APLICACIÓN

En este apartado se explica cómo está estructurada la aplicación, de manera que se permite examinar todas las funcionalidades que nos aporta el usar Istio. Para ello, se implementa una aplicación basada en microservicios que se le añade las capas de Kubernetes e Istio.

Esta aplicación está compuesta por siete microservicios en distintos lenguajes de programación. Lo que nos permite demostrar los beneficios de usar este tipo de arquitectura y crear las conexiones entre ellos.

Una vez que se tenga implementado todos los microservicios y añadidos a los contenedores se le añade la capa de Kubernetes lo que nos va a permitir orquestar los distintos contenedores, así como la administración y escalado de los mismos. Luego se le añade la capa de malla de servicios que nos permite controlar el tráfico, seguridad y trazabilidad.

La aplicación se desplegará en la plataforma de Google Cloud que nos permite acceder a ella desde cualquier lugar.

8.1. FRONT-END

La aplicación tendrá un servicio Home, realizado con PHP, HTML5 y CSS, el cual estará encargado de mostrar la página web y llamar a los distintos microservicios.

El diseño de la web se ha hecho de manera sencilla para permitir realizar operaciones CRUD (*Create, Read, Update and Delete*) sobre la base de datos. Así mismo también probar funcionalidades de control del tráfico.

La principal funcionalidad de la web es realizar peticiones *post*, *get*, *put* y *delete* permitiendo mostrar los resultados obtenidos.

8.2. BACK-END

A continuación, se explica cómo esta implementado el *back-end* de la aplicación, sus servicios y base de datos.

8.2.1. DESCRIPCIÓN DE LA APLICACIÓN

Los servicios implementados formaran la arquitectura de microservicios. Cuyo objetivo es visualizar la información de las llamadas y mostrarlo en la página web Home.

Para realizar estas peticiones, el servicio Home llama a los servicios mediante peticiones HTTP y el resultado lo muestra en la web. La respuesta de llamar a los servicios se proporciona en formato JSON.

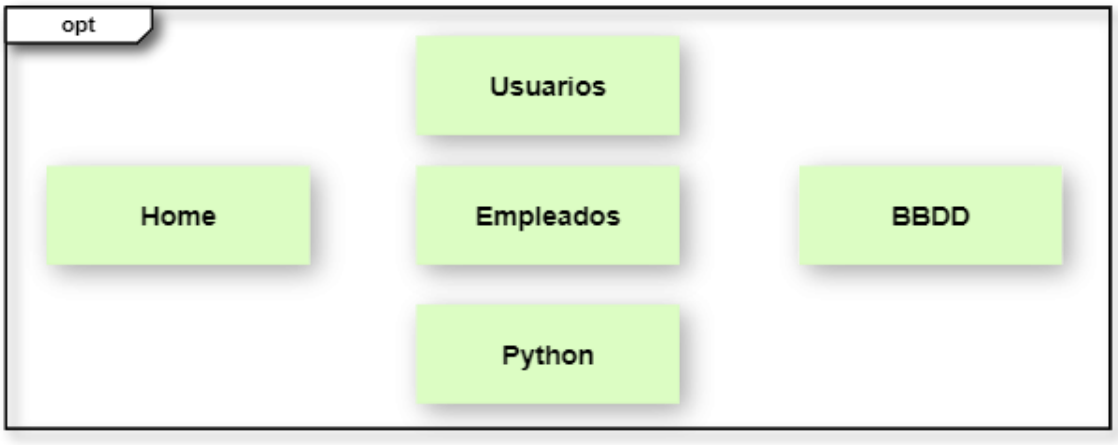


Figura 28- Diagrama Microservicios

Las funciones de los microservicios son las siguientes:

- **Home:** Es la página web, que se encargara de hacer las peticiones a los microservicios. Esta desarrollada en PHP, HTML5 y CSS.
- **Usuarios:** Es una API REST encargada de devolver la información de los usuarios en formato JSON.
- **Empleados:** Es una API REST encargada de devolver la información relacionada con los empleados, por ejemplo, su cargo.
- **Python:** Es una API REST encargada de devolver un listado de las edades y DNI de los usuarios en formato JSON.
- **BBDD:** Es la base de datos que permite la consulta de las tablas usuario y empleado.

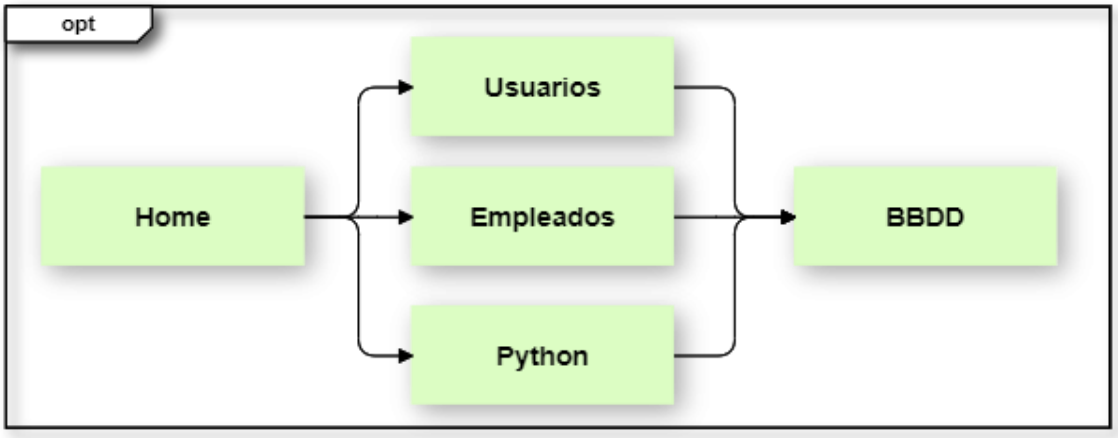


Figura 29- Diagrama de conexiones entre microservicios

El flujo de las peticiones será el marcado por el servicio Home, que realizará peticiones a los servicios Usuarios, Empleados y Python. Estos servicios llamaran al servicio BBDD y este devolverá la información al interfaz en formato JSON.

El servicio Python y Empleados tendrá dos versiones, que nos va a permitir probar las distintas funcionalidades de tráfico y enrutamiento. Para comprobar que los servicios estén operativos, la interfaz realizará peticiones y verificará que se puede conectar a ellos, con el objetivo de saber si están activos.

Por lo tanto, la arquitectura de la aplicación tiene unos siete microservicios y varias versiones de estos. Se le va a añadir la capa de malla de servicios para explotar las funcionalidades que nos ofrece.

La comunicación entre los servicios será mediante el protocolo REST que utiliza el formato JSON para trasportar la información.

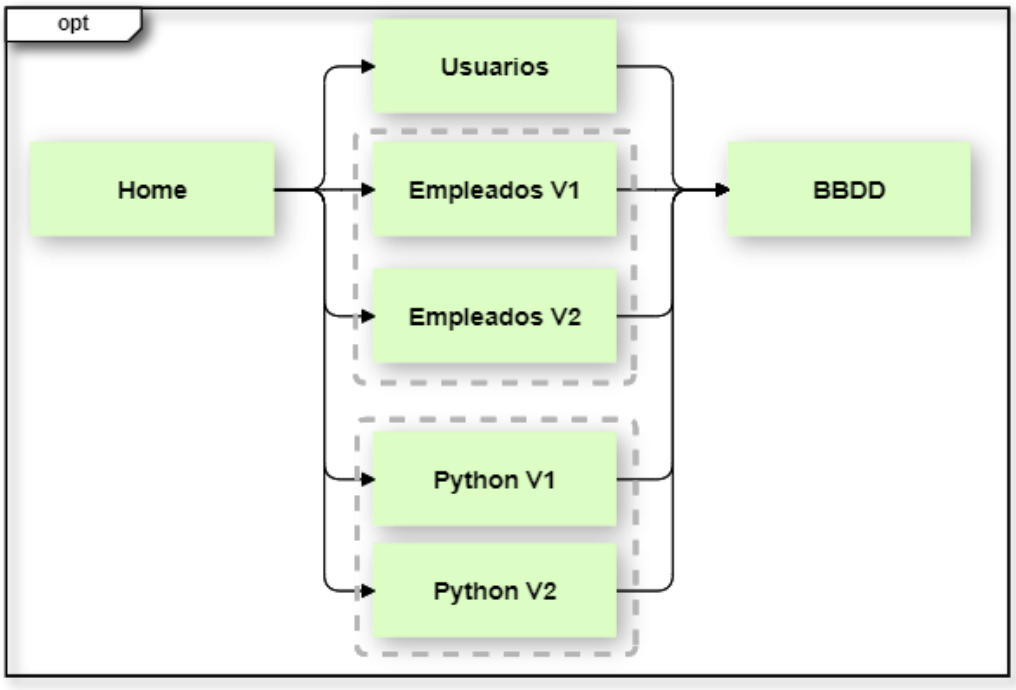


Figura 30- Diagrama de microservicios con versiones

Hemos expuesto de forma breve la estructura de la aplicación implementada en este proyecto. En los próximos puntos vamos a tratar el diseño de esta, de forma más profunda.

Como se puede observar esta aplicación ha sido diseñada y desarrollada siguiendo varias fases, comenzando con la arquitectura de microservicios y escalándola hacia Docker, Kubernetes e Istio. Este es el orden que vamos a seguir en la explicación de esta y la siguiente sección.

8.2.2. BASE DE DATOS

Para la base de datos se usa un sistema de gestión MySQL, ya que es de código abierto y tiene una arquitectura cliente-servidor. Es relacional, y esto nos facilita el control en la inserción de datos. Puesto que el objetivo es comprobar el funcionamiento de la malla de servicio, hemos diseñado una base de datos sencilla y de fácil uso, de igual manera que los demás servicios.

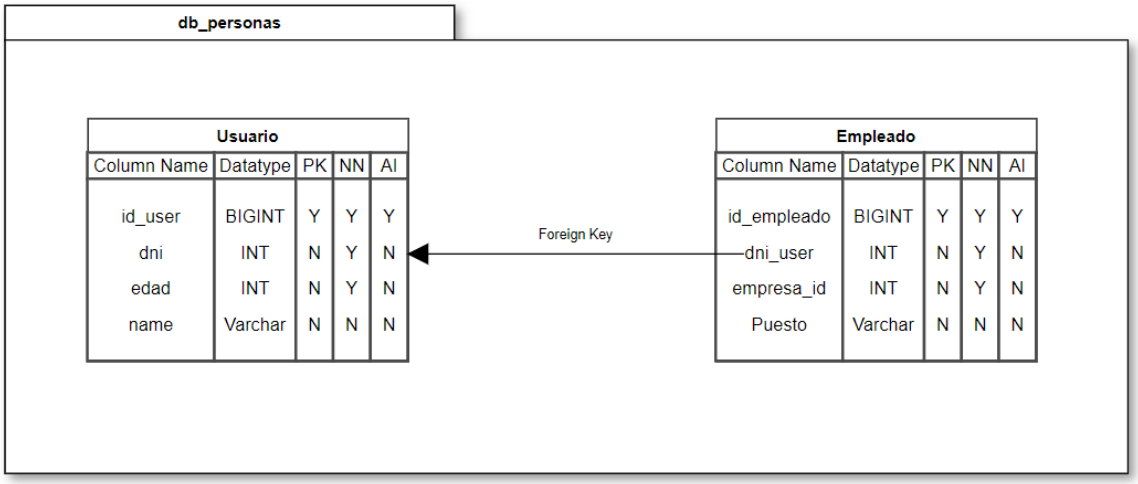


Figura 31- Diagrama UML de la base de datos

Como podemos observar en el diagrama UML la base de datos está compuesta por dos tablas: “Empleado” y “Usuario”. Cada una poseen cuatro atributos: un identificador, dos enteros y una cadena *varchar* de 256B como longitud máxima. Las columnas están divididas en el nombre de cada atributo (“Column Name”), su tipo (“Datatype”), si es clave primaria (“PK”), si no puede ser vacío (“NN”), y si es auto incremental (“AI”).

Además, la tabla “Empleado” tiene definida una *Foreign Key*, que nos permite controlar que ningún dato insertado en esta tabla, no dependa de otro ya insertado en la tabla “Usuario”.

8.2.3. SERVICIO USUARIO

Es el primer servicio API REST diseñado e implementado, se conecta a la base de datos anteriormente descrita. El diagrama de clases que debe seguir es el siguiente:

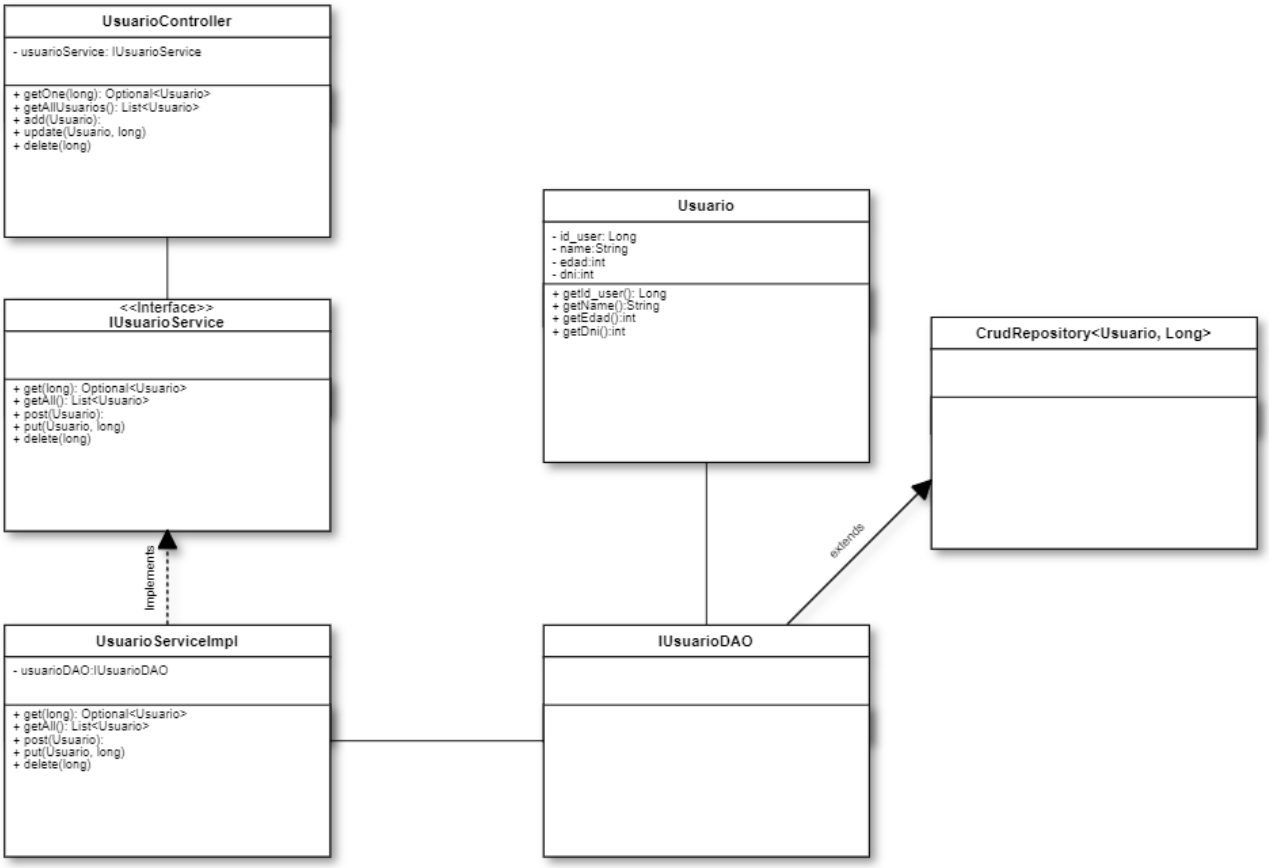


Figura 32- Diagrama de clases del servicio Usuario

Como podemos observar, está formado por un modelo de datos "Usuario" con los mismos atributos que posee la tabla de la base de datos que da nombre a esta clase. Además, tiene una interfaz que implementa el patrón DAO del que hablaremos en los puntos siguientes. Esta interfaz extiende la clase "CrudRepository" que nos proporciona todas las funciones necesarias para desarrollar las *CRUD*. La interfaz DAO es utilizada por la clase que implementa otra interfaz que contiene las funciones del servicio. Por último, tenemos una clase *Controller* que contiene las rutas de la API.

8.2.4. EMPLEADO

Este servicio posee una estructura similar al anterior, solo que esta vez trata los datos de la tabla “Empleado”.

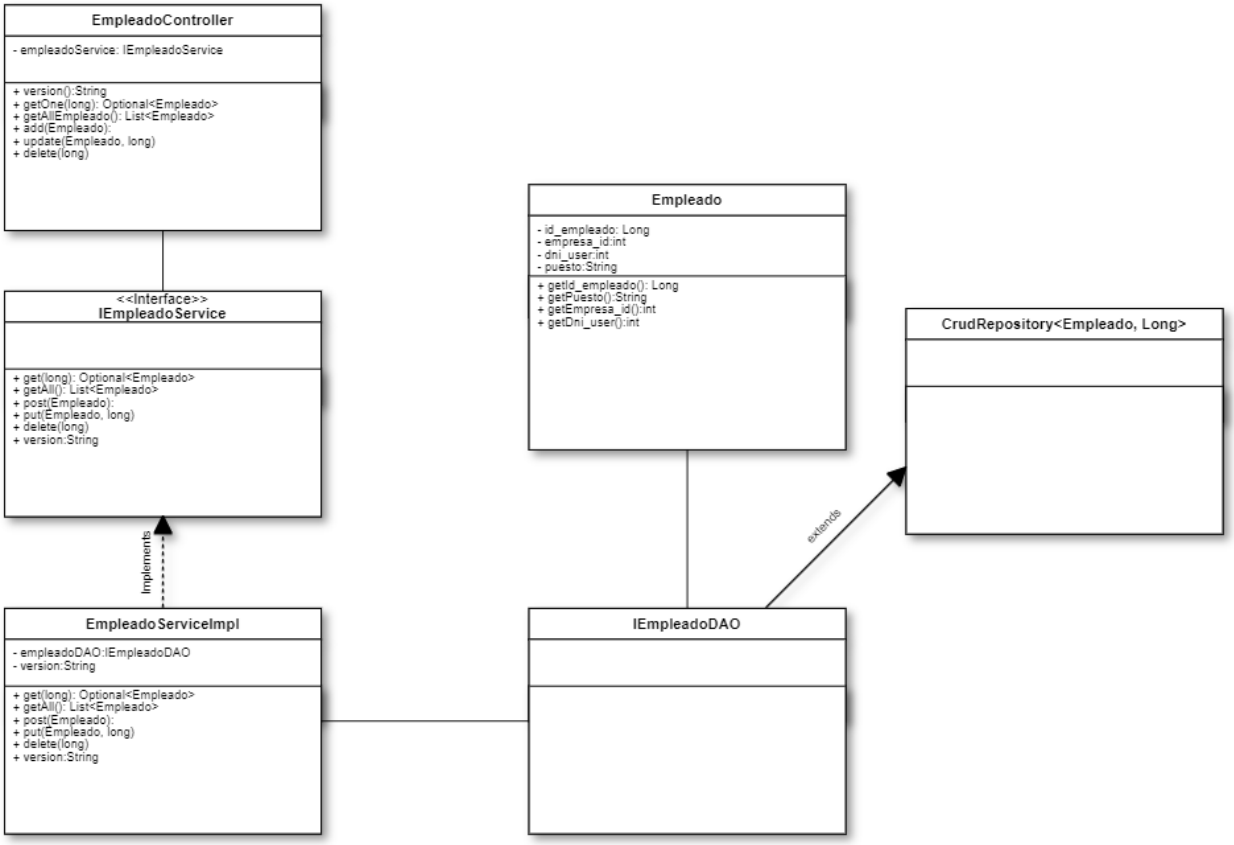


Figura 33- Diagrama de clases del servicio Empleado

Las únicas diferencias con el diagrama de clases anterior son:

- Clase “EmpleadoServiceImpl” posee un atributo utilizado para saber la versión del servicio utilizado, como expusimos en el punto siete, el servicio “Empleado” tiene dos versiones y este atributo es el que nos permite saber a cuál de las dos estamos accediendo.
- Clase “IEmpleadoService” y “EmpleadoController” el método “versión”.

La adición de una de las versiones de este servicio puede parecer inútil, pero la hemos implementado con el objetivo de comprobar el funcionamiento de la malla de servicios, puesto que van a trabajar en paralelo y utilizados para distintas pruebas en el control de tráfico, como puede ser el balanceo de carga.

8.2.5. PYTHON

Este servicio es el único implementado en Python, también se conecta a la BBDD, aunque esta vez solamente podrá leer los datos, y no modificarlos. El diagrama de clases de este servicio es el siguiente:

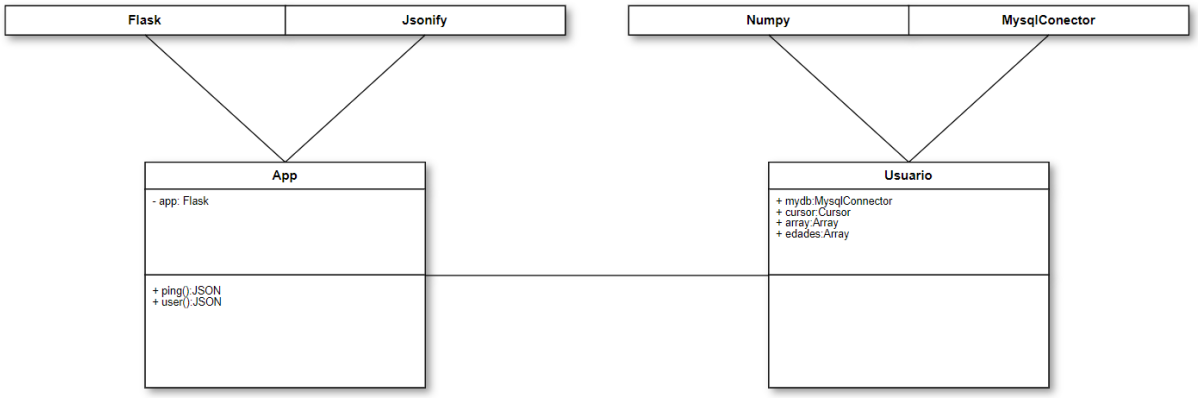


Figura 34- Diagrama de clases del servicio Python

Como se puede observar, este servicio es muy simple, posee dos clases que a su vez hacen uso de dos librerías diferentes. La clase "Usuario" es la que accede a la BBDD obteniendo un array con las edades de los usuarios, para ello utiliza las librerías "Numpy" y "MysqlConnector". La clase "App" contiene las rutas de la API, mostrando un JSON con las edades que obtiene de la clase "Usuario", para ello utiliza las librerías "Flask" y "Jsonify".

La segunda versión de este servicio añade una funcionalidad a la API REST, se añade un atributo a la clase "Usuario" que utiliza para leer los DNIs de los usuarios. Además, se le añade una ruta a "App" para desempeñar esta nueva funcionalidad.

En este caso, las versiones de este servicio se han diseñado con el objetivo de comprobar el control de versiones que nos proporciona Istio.

8.3. DOCKER

El segundo paso en el diseño e implementación de este proyecto es una vez que tenemos la arquitectura inicial de microservicios, establecer el diseño de los contenedores en Docker, en la siguiente figura se puede observar el número de contenedores necesarios y el sistema operativo que se ejecuta en esos contenedores:

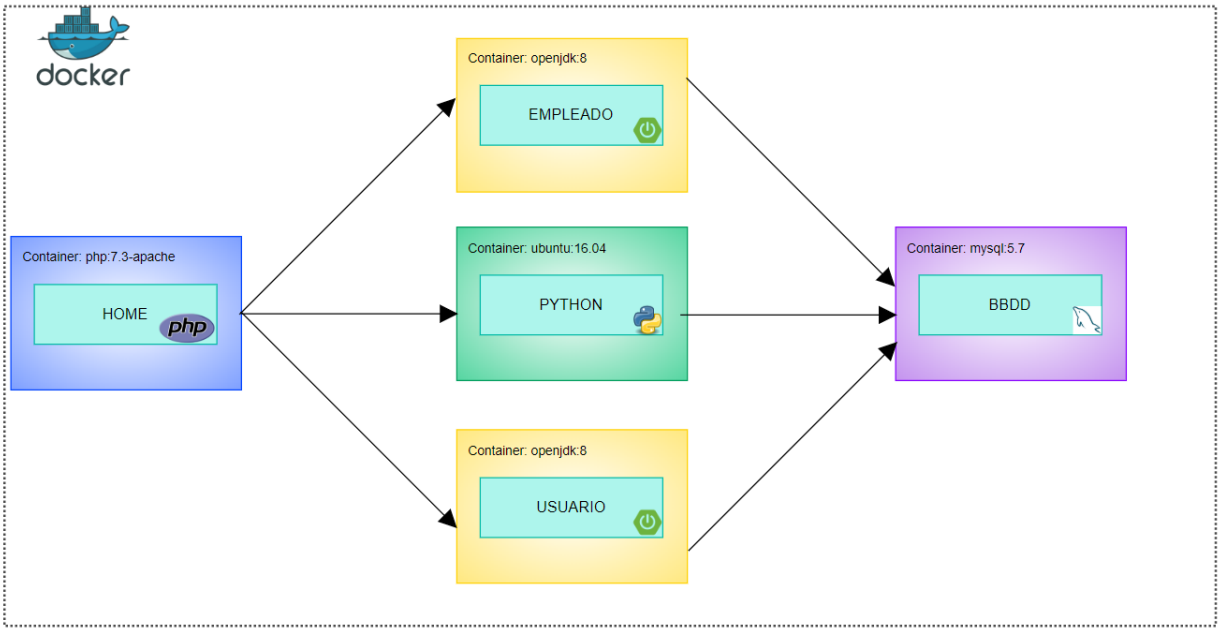


Figura 35- Diagrama del diseño de la arquitectura de Docker

Como se puede observar los servicios “Usuario” y “Empleado” están ejecutados sobre el mismo sistema operativo, pero en distintos contenedores, esto se debe a que en un futuro podemos establecer conexiones diferentes en estos dos servicios, además nos facilita el control de fallos, sobre estos servicios.

A continuación, vamos a exponer los recursos necesarios para la implementación de la arquitectura en Docker:

Recurso	Descripción
DockerFile	Hemos definido una serie de Dockerfile por cada uno de los servicios diseñados.
Imágenes	Necesitamos imágenes base para poder ejecutar los microservicios que se usan las de openjdk, Ubuntu y mysql. Se va a obtener y almacenar una imagen por cada servicio y versión.
Contenedores	Se ejecutarán las imágenes de los microservicios para tener los contenedores usando el comando *docker run*.
Volúmenes	Es una versión de los volúmenes ya explicados en el contexto. Este recurso es una forma de almacenamiento en local que tienen un ciclo independiente al de un *contenedor*. Hemos utilizado una unidad de este recurso para mantener los datos almacenados.

Network	Se define una red network tipo puente para todos los contenedores lo que permite que se puedan comunicar entre sí, facilitándonos esta configuración, ya que de otra manera tendríamos que establecer más *links* (ya explicados en el contexto).

Tabla 3- Recursos en el diseño de Docker

8.4. KUBERNETES

Una vez que hemos expuesto el diseño que seguimos en Docker, pasamos a hablar del diseño en Kubernetes. Antes de comenzar a exponer los recursos necesarios para la implementación, debemos mostrar la estructura de nuestra aplicación en Kubernetes:

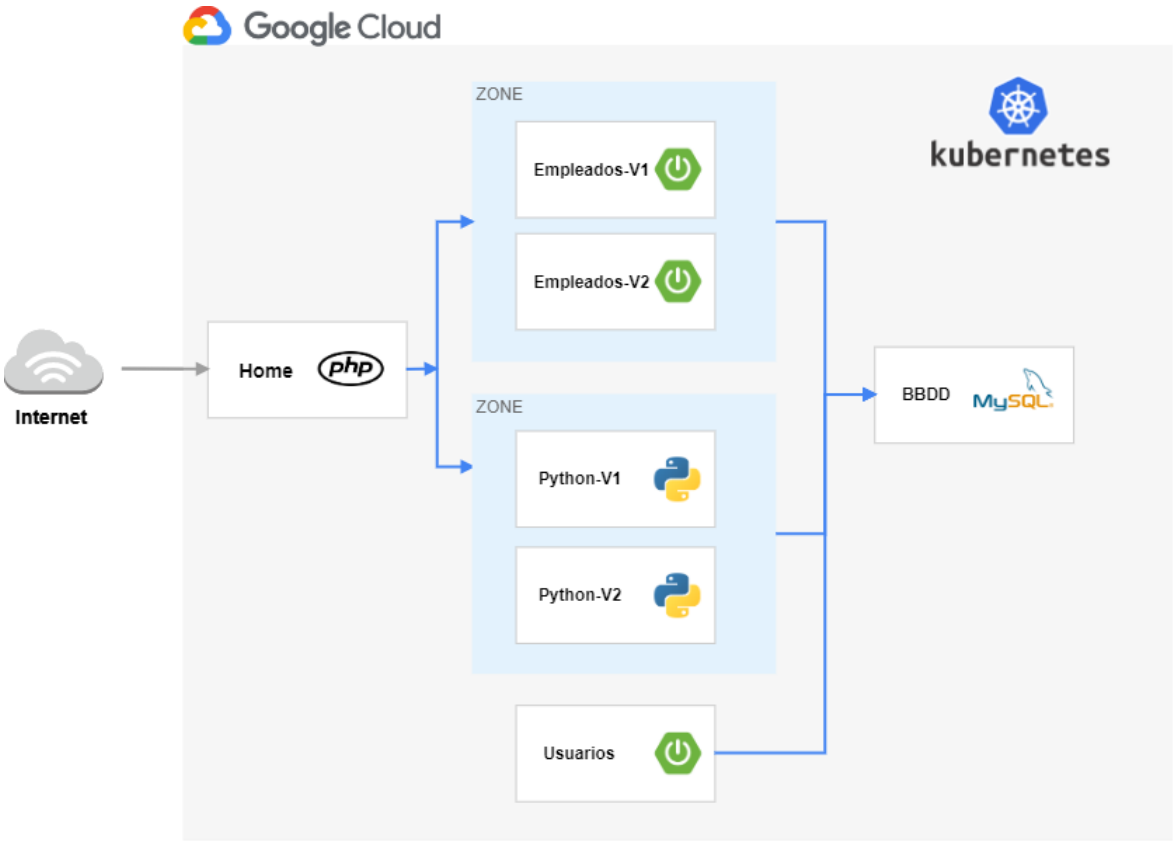


Figura 36- Diagrama del diseño de la arquitectura en Kubernetes

Como se puede ver en la imagen, hay diferentes servicios y los distintos lenguajes de programación en los que están implementados. Así mismo, se visualiza las distintas versiones y conexiones de estos.

El despliegue de la aplicación es mediante Google Cloud utilizando como orquestador a Kubernetes. Lo que nos permite tener un entorno de administración, así como organizar los recursos que queremos.

Para el despliegue de la aplicación hemos definido los siguientes recursos, que son estrictamente necesarios para el correcto funcionamiento de la aplicación:

Recurso	Descripción
Deployment	Hemos definido un *deployment* por cada uno de los servicios, teniendo en cuenta que las versiones también necesitan de este recurso. Es utilizado con el objetivo de controlar las instancias de los *pods* y sus contenedores.
Service	Necesitamos de una unidad de este recurso por cada uno de los servicios implementados, pero en este caso solo necesitaremos de un *service* para las distintas versiones, ya que el *label* que define el nombre de la aplicación y al que se dirige el *service* es el mismo independientemente de la versión. Es utilizado para que se pueda establecer la conexión entre los diferentes servicios.
Pod	Viene definido en los archivos de los *deployments*, y vamos a tener uno por cada *deployment*. Como veremos en el siguiente apartado en Istio aumentarán a dos, ya que el *proxy* Envoy necesitará uno independiente.
Replica Controller	Al igual que los *pods*, vienen definidos en el archivo de los *deployments*, con el objetivo de controlar las instancias de estos.
Label	Como ya adelantamos, utilizamos esta potente herramienta en los *deployments* y *services*, con el objetivo de identificar y seleccionar correctamente los recursos.

Tabla 4- Recursos en el diseño de Kubernetes

Con los recursos vistos anteriormente podríamos levantar la aplicación de forma correcta, pero hemos de añadir algunos recursos más, con el objetivo de aprovechar de mejor forma las funcionalidades de Kubernetes, además nos facilitarán el proceso de desarrollo y producción.

Recurso	Descripción
Secret	Hemos definido un *secret* para la base de datos, ya que tenemos información sensible como es la contraseña para acceder a esta.
Persistent Volume	Es una versión de los volúmenes ya explicados en el contexto. Este recurso es una forma de almacenamiento en el clúster que tienen un ciclo independiente al de un *pod*. Hemos utilizado una unidad de este recurso para mantener los datos almacenados.
Persistent Claim Volume	De la misma manera que el anterior recurso es una versión de los volúmenes de Kubernetes. Funcionan como una solicitud de almacenamiento por parte del usuario. Hemos utilizado un *persistent claim volume* para la correcta interacción con la base de datos.
Namespaces	Vamos a establecer un clúster virtual para el despliegue de nuestra aplicación.

Tabla 5- Recursos adicionales en el diseño de Kubernetes

8.5. ISTIO

La capa de malla de servicios no es una arquitectura nueva, sino la capa estructural básica que aplicamos a la arquitectura de microservicios. Lo que se hará es añadir esa capa a la arquitectura anterior y agregaremos un proxy para cada servicio por donde pasará toda la comunicación de los servicios.

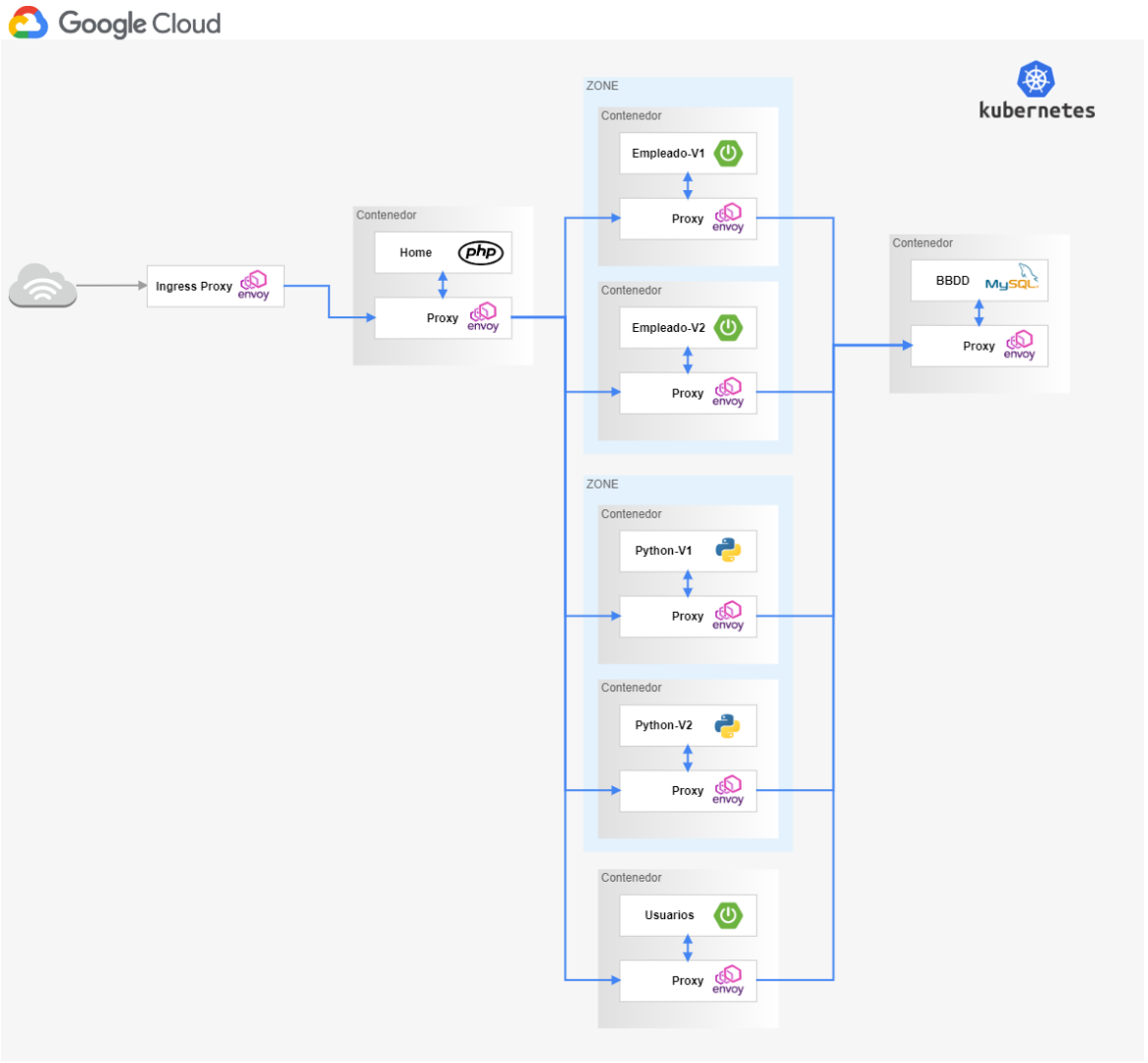


Figura 37- Diagrama del diseño de Istio

Como se visualiza en la imagen anterior, lo que se modifica en la arquitectura es que se incluye el proxy Envoy a cada uno de los microservicios, donde Envoy es la tecnología responsable de implementar el proxy. El resto es igual que la arquitectura, pero la diferencia es que ahora tenemos la capa de Istio.

Para el despliegue de la aplicación se han definido los siguientes recursos, que son estrictamente necesarios para el correcto funcionamiento de la aplicación:

Recurso	Descripción
Virtual services	Hemos definido una serie de virtual service que son los bloques de construcción clave de la funcionalidad de enrutamiento de tráfico de Istio. Un servicio virtual nos permite configurar cómo se enrutan

	las solicitudes a un servicio dentro de una malla de servicios, basándose en la conectividad básica.
Destination rules	Necesitamos de los DestinationRules junto con los servicios virtuales, ya que las reglas de destino son una parte clave de la funcionalidad de enrutamiento de tráfico de Istio. Estas reglas de destino se usan para configurar lo que sucede con el tráfico para ese destino. Se aplican después de evaluar las reglas de enrutamiento del servicio virtual, por lo que se aplican al destino "real" del tráfico.
Gateways	Se utiliza el Gateway para administrar el tráfico entrante y saliente de la malla, lo que le permite especificar en qué tráfico desea ingresar o salir de la malla. Las configuraciones del gateway se aplican a los servidores proxy Envoy independientes que se ejecutan en el borde de la malla, en lugar de los servidores proxy Envoy que se ejecutan junto con las cargas de trabajo de servicio.
Service entries	Se utiliza un service entries para agregar una entrada al registro de servicio que Istio mantiene internamente. Después de agregar la entrada de servicio, los servidores proxy de Envoy pueden enviar tráfico al servicio como si fuera un servicio en su malla. La configuración de entradas de servicio le permite administrar el tráfico de los servicios que se ejecutan fuera de la malla, incluidas las siguientes tareas
Sidecars	Se utiliza el patrón sidecar en cada microservicio para la comunicación entre ellos

Tabla 6- Recursos en el diseño de Istio.

8.6. PATRONES DEL DISEÑO

A continuación, se explican los patrones usados en el proyecto.

8.6.1. SIDECAR

Inyección de un componente en un proceso o contenedor independientes, proporcionando encapsulación y aislamiento. En este caso, se utiliza en la inserción del *proxy* Envoy para las funcionalidades de Istio, como ya hemos visto en los puntos anteriores.

Un sidecar es un contenedor que funciona como punto de entrada en el *pod* y su propósito es dar soporte al contenedor principal. En resumen, el sidecar posibilita el control del tráfico en la malla de servicios, interfiriendo en el plano de datos, y siendo dirigido por el plano de control.

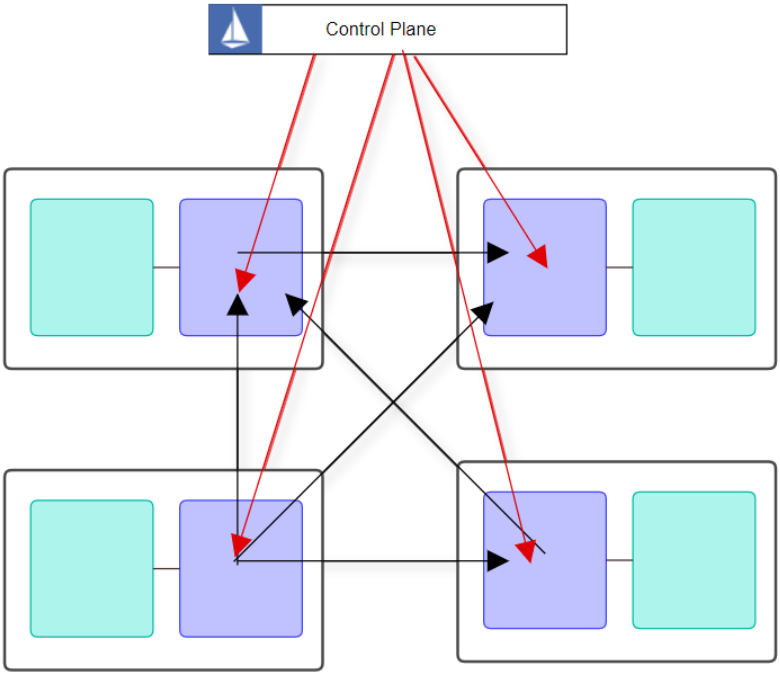


Figura 38- Diagrama del patrón sidecar

Como se aprecia en la figura el plano de control supervisa y guía a los sidecares, en este caso en azul, y son estos los que se comunican con el contenedor principal y con los demás sidecares.

8.6.2. DAO

El patrón DAO (*Data Access Object*) separa la lógica de negocio del acceso a la base de datos nos ayuda a solucionar el problema de la gestión de diversas fuentes de datos y abstrae la forma con la que se accede a estos datos.

En este caso lo hemos utilizado en los servicios “Empleado” y “Usuario”, en el acceso a la base de datos SQL.

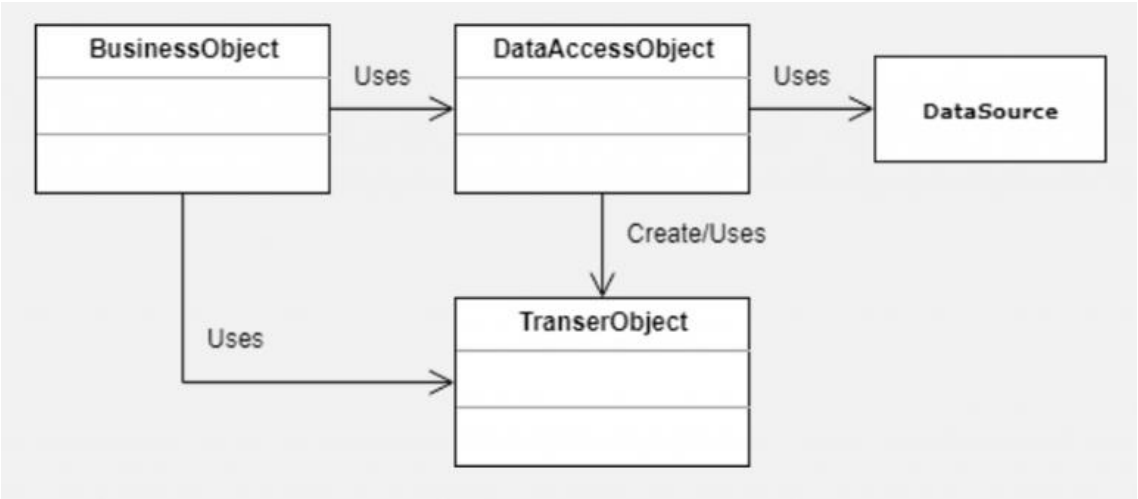


Figura 39- Diagrama del patrón DAO
Fuente: https://www.oscarblancarteblog.com/2018/12/10/data-access-object-dao-pattern/

En este caso este diagrama resume la funcionalidad del patrón DAO general, pero en este caso, el objeto “TransferObject”, no es necesario.

8.6.3. MVC

En este proyecto, se utiliza una versión reducida del patrón modelo-vista-controlador, ya que, al no necesitar un interfaz gráfico, solo disponemos de una clase que realiza las funciones de un controlador, y otra que desempeña las de modelo.

Al igual que el patrón anterior, se implementan en los servicios “Empleado” e “Usuario”. El controlador es la clase que posee las rutas de la API REST y el modelo es la clase que contiene los atributos y métodos para interactuar y acceder a los datos.

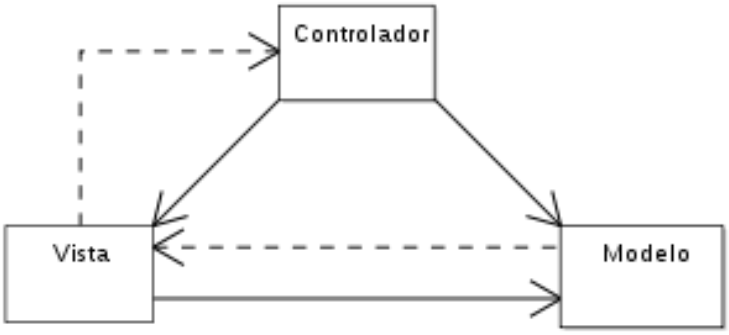


Figura 40- Diagrama del patrón MVC
Fuente: https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

8.6.4. SAGA

El patrón Saga nos ayuda a garantizar que las transacciones distribuidas locales funcionen correctamente. En este proyecto se implementa la versión de este patrón que se desarrolla con orquestadores.

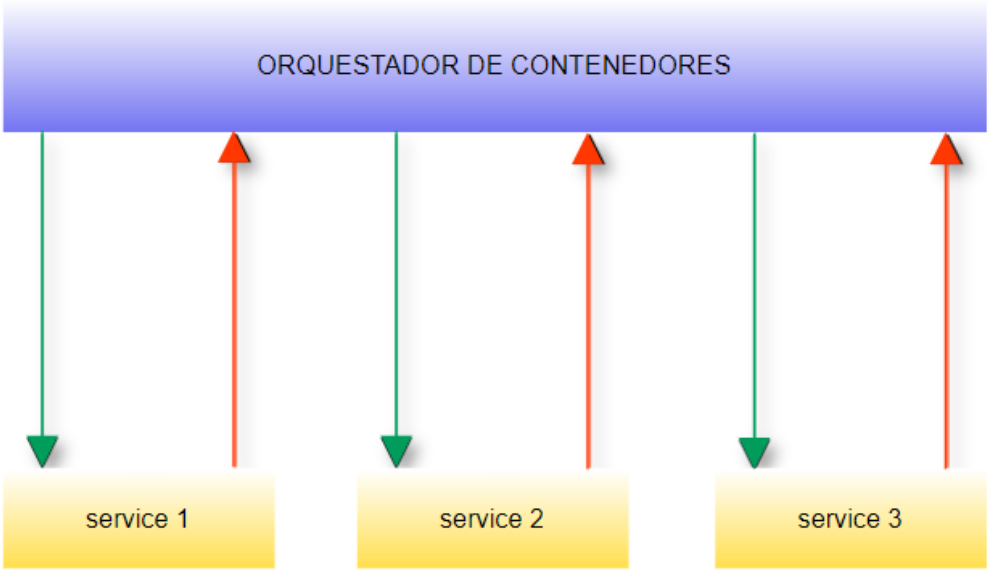


Figura 41- Diagrama del patrón Saga

Si alguna transacción falla el orquestador es el responsable del *rollback* de las transacciones previas ejecutadas. Esto es posible gracias a los recursos y funcionalidades de Kubernetes.

9. IMPLEMENTACIÓN

En esta sección vamos a exponer la implementación de este proyecto siguiendo el orden del diseño. Comenzamos con la arquitectura de microservicios inicial y terminaremos con la malla de servicios.

9.1. ARQUITECTURA DE MICROSERVICIOS

9.1.1. BASE DE DATOS

Iniciamos la implementación de este proyecto con la base de datos SQL, ya expuesta en el diseño. El desarrollo de esta parte se ha realizado con la herramienta MySQL Workbench, como ya adelantamos anteriormente, pero debido a que queremos exponer un proceso de desarrollo general, es decir, sin el uso de los softwares que no sean estrictamente necesarios, vamos a mostrar la implementación de los archivos con formato SQL, en un editor de texto.

Para nuestra base de datos hemos implementado dos tablas, y sendos archivos con el objetivo de mejorar su legibilidad. El primer archivo contiene las instrucciones que borran las tablas si ya existen, para que no se produzcan errores, y la definición de la tabla "usuario", además de los datos que insertamos para que nos este vacía. El segundo archivo posee la estructura de la tabla "empleado" y los datos que insertamos en esta.

```
DROP TABLE IF EXISTS `empleado`;
DROP TABLE IF EXISTS `usuario`;

CREATE TABLE `usuario` (
  `id_user` int NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `dni` int NOT NULL,
  `edad` int NOT NULL,
  PRIMARY KEY (`id_user`),
  UNIQUE KEY `dni_UNIQUE` (`dni`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8mb4;

INSERT INTO `usuario` VALUES (1,'Raul',5784965,21),(2,'Pedro',5796548,45),
(3,'Luis',68593741,12),(4,'Laura',87563215,39),(5,'Isabel',78326142,43),
(6,'Alvaro',96374861,24),(7,'Pablo',12489675,25),(8,'Loren',124898795,25);
```

Figura 42- Archivo SQL para la tabla "usuario"

Como ya expusimos en el diseño esta tabla tiene cuatro atributos, uno de ellos, el identificador, es auto incremental, además es la clave primaria de la tabla.

```
CREATE TABLE `empleado` (
  `id_empleado` int NOT NULL AUTO_INCREMENT,
  `empresa_id` int NOT NULL,
  `dni_user` int NOT NULL,
  `puesto` varchar(45) NOT NULL,
  PRIMARY KEY (`id_empleado`),
  KEY `fk_empleado_usuario_idx` (`dni_user`),
  CONSTRAINT `fk_empleado_usuario` FOREIGN KEY (`dni_user`) REFERENCES `usuario` (`dni`) ON DELETE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8mb4;

INSERT INTO `empleado` VALUES (1,1,5784965,'Presidente'),(2,1,5796548,'Vice Presidente'),
(3,1,78326142,'Delegado de Marketing'),(4,2,96374861,'Contable'),(6,3,68593741,'Gerente');
```

Figura 43- Archivo SQL para la tabla "empleado"

La única diferencia de esta tabla, además del cambio de nombres, es que posee una clave foránea que establece una relación de esta tabla con la de “usuario”, definiendo que en caso de eliminar algún dato de la tabla “usuario”, se borraría también su referencia en la tabla “empleado”.

9.1.2. API REST

SPRING BOOT

La primera parte de la implementación de este proyecto fueron las APIs REST, en concreto con los servicios que se implementaron en Spring Boot.

La herramienta utilizada para ello, como ya adelantamos ha sido IntelliJ IDEA. Esta herramienta nos ofrece una manera rápida y cómoda para crear un proyecto en Spring Boot, debemos seleccionar la opción de “Create a New Project”, y aparecerá la siguiente pantalla:

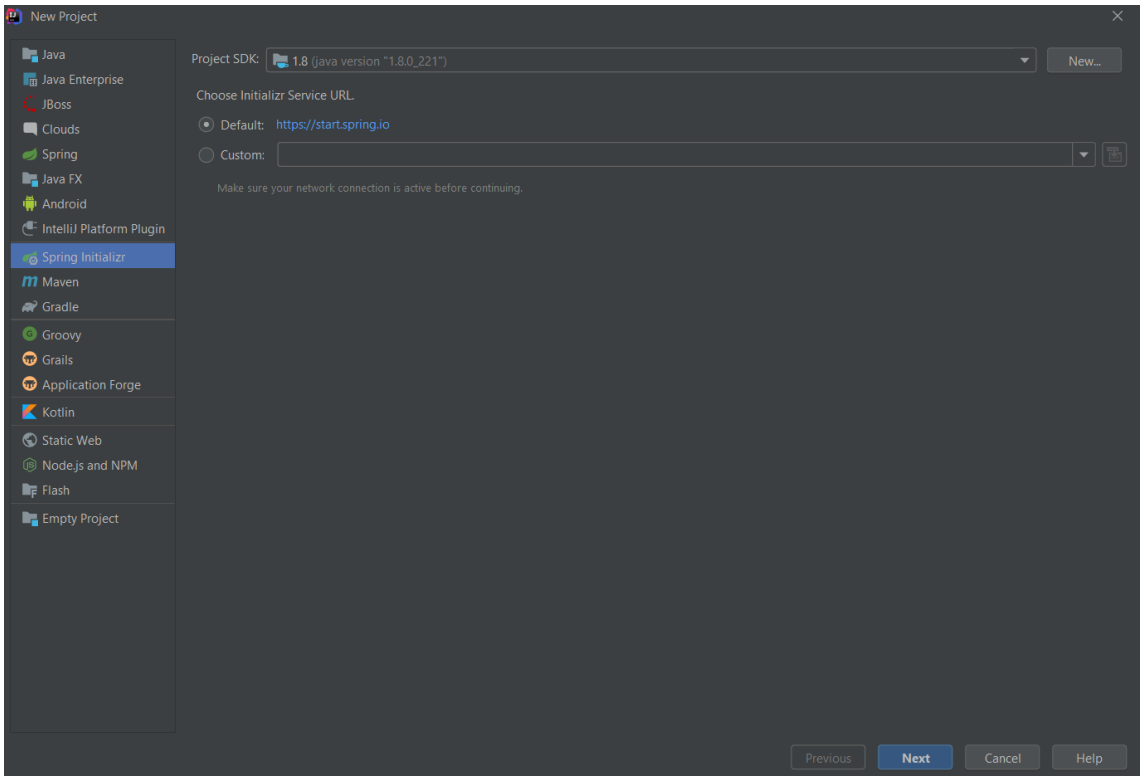


Figura 44- Creación del proyecto en Spring Boot

Donde deberemos seleccionar “Spring Initializr” en la columna lateral. En este momento tenemos dos opciones o presionar sobre el botón inferior izquierdo, para seguir con la creación del proyecto en IntelliJ IDEA o pulsar sobre la URL que aparece en la imagen. Con el objetivo de hacerlo de forma universal para cualquier software vamos a mostrar la opción de creación a través de la URL. Una vez presionamos sobre el hiperenlace, nos aparece la siguiente pantalla:

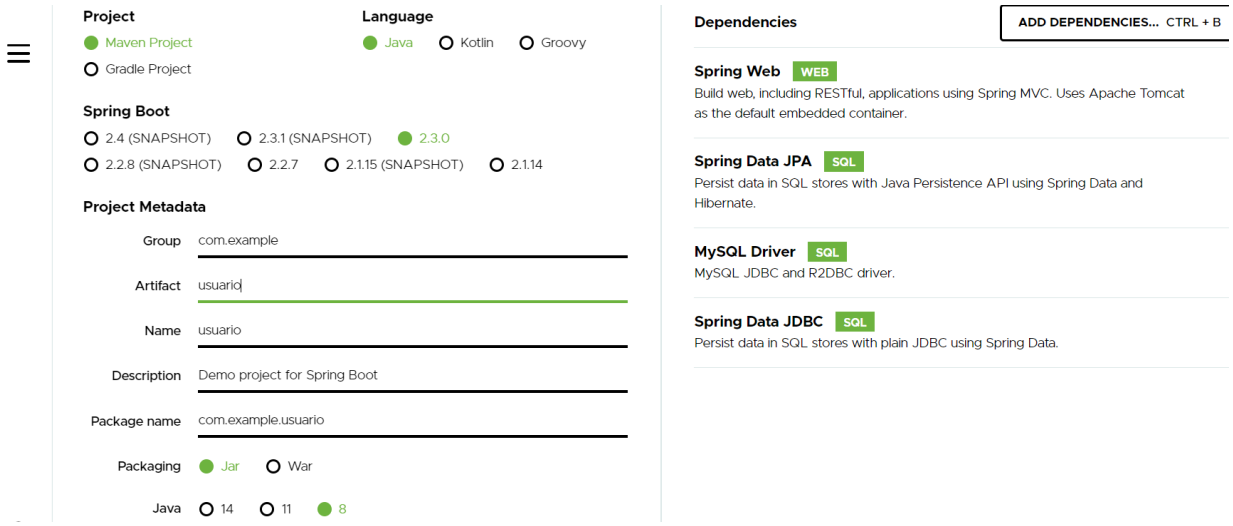


Figura 45- Parámetros del proyecto en Spring Boot

En este caso vamos a seleccionar el nombre “usuario” como ejemplo, para no tener que repetir el mismo proceso con el servicio “Empleado”. Seleccionamos Java como lenguaje base, la versión de Spring Boot y la versión de Java que en este caso es la octava. A la derecha vemos una pestaña con las dependencias del proyecto necesarias, de esta manera Spring las añadirá directamente al archivo pom.xml. En este caso hemos elegido estas dependencias iniciales, aunque con el transcurso del desarrollo pueden ser modificadas.

Una vez creado el proyecto, lo abriremos con IntelliJ IDEA, y comprobaremos que se han añadido las dependencias correctamente en el archivo pom.xml. En el caso del servicio “Usuario” y “Empleado” tenemos las siguientes dependencias:

- spring-boot-starter-data-jdbc: Para el acceso a datos relacionales con Spring
- spring-boot-starter-data-jpa: Almacenamiento y recuperación de datos relacionales.
- spring-boot-starter-web: Construcción de proyectos webs
- spring-boot-devtools: Paquete básico para la utilización de Spring
- mysql-connector-java: Para la conexión con la base de datos SQL.
- spring-boot-starter-test: Paquete básico para realizar test.
- junit-vintage-engine: Paquete básico para el uso de Junit.
- spring-boot-maven-plugin: Paquete básico para la utilización de Maven en el proyecto.

El siguiente paso es la propia implementación del proyecto en este caso vamos a utilizar el código del servicio “Usuario”, ya que “Empleado” es bastante parecido. La estructura de clases quedaría de la siguiente manera:

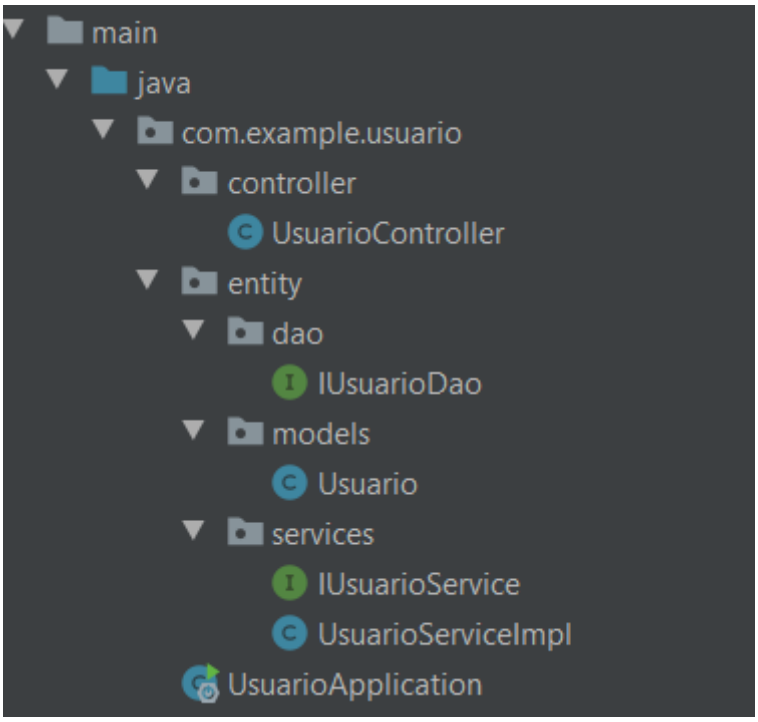


Figura 46- Creación proyecto Spring Boot

Como se puede observar, el proyecto está dividido en dos paquetes principales, "Controller" y "Entity", está división sigue los patrones ya explicados, de manera que sea más rápido entender el código.

El paquete "Entity" está dividido a su vez en otros tres paquetes: "DAO", "Models" y "Services". La primera clase que debemos implementar es "usuario", dado que contiene el modelo de datos para comunicarse con la tabla de la base de datos "Usuario". Esta clase posee los siguientes identificadores y atributos:

```
@Entity
@Table(name = "usuario")
public class Usuario {
```

Figura 47- Implementación del servicio usuario (1)

@Entity identifica a la clase como una entidad y @Table identifica que es el modelo de la tabla "usuario" de la base de datos

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id_user;
@NotEmpty
private String name;
@NotNull
private int edad;
@NotNull
private int dni;
```

Figura 48- Implementación del servicio usuario (2)

Las funciones de los identificadores siguen la implementación de la base de datos, estas son:

- @Id: Atribuye la cualidad de identificador al atributo.
- @GeneratedValue: genera los identificadores.
- @NotEmpty: Especifica que la cadena no puede ser vacía.
- @NotNull: Especifica que el entero no puede ser nulo.

Por último, en esta clase, hacen falta los siguientes métodos:

```java
public Usuario() { super(); }

public Usuario(@NotEmpty String name, @NotNull int edad, @NotNull int dni) {
    this.name = name;
    this.edad = edad;
    this.dni = dni;
}

public Long getId_user() { return id_user; }

public String getName() { return name; }

public int getEdad() { return edad; }

public int getDni() { return dni; }

public void setId_user(Long id_user) { this.id_user = id_user; }
```

Figura 49- Implementación del servicio usuario (3)

Se reducen a dos constructores de la clase, a los métodos Get de cada atributo y al método Set del atributo Id utilizado en el método encargado de actualizar la tabla.

Una vez que hemos terminado la implementación del modelo, debemos implementar la clase (en este caso interfaz) necesaria para el desarrollo del patrón DAO. Al extender a la clase “CrudRepository” la implementación de la clase es sencilla.

```
public interface IUsuarioDao extends CrudRepository<Usuario, Long> {
}
```

Figura 50- Implementación del servicio usuario (4)

Para finalizar con la implementación del paquete "Entity" debemos desarrollar las dos clases del paquete "Service", una interfaz y la clase que la implementa. Con el objetivo de no repetir código vamos a mostrar solamente el de la clase que implementa la interfaz, "UsuarioServiceImpl". En este caso posee los siguientes identificadores y atributos:

```
@Service
public class UsuarioServiceImpl implements IUsuarioService{

    @Autowired
    private IUsuarioDao usuarioDao;
```

Figura 51- Implementación del servicio usuario (5)

El identificador @Service le atribuye la cualidad de servicio a la clase, y @Autowired especifica que al crearse una instancia de "UsuarioServiceImpl" se debe inyectar en el atributo "usuarioDao" las dependencias e información de la clase DAO.

Los métodos necesarios en esta clase, implementas las funciones básicas del CRUD (explicado en el contexto), de esta manera tenemos las siguientes funciones:

- *Create*:

```
@Override
public void post(Usuario usuario) { usuarioDao.save(usuario); }
```

Figura 52- Implementación del servicio usuario (6)

Inserta una fila en la tabla de la base de datos, con la información guardada en el atributo "usuario".

- *Read*:

```
@Override
public Optional<Usuario> get(long id_user) { return usuarioDao.findById(id_user); }
```

Figura 53- Implementación del servicio usuario (7)

Este método obtiene el JSON que contiene la información del usuario con identificador "id_user".

```
@Override
public List<Usuario> getAll() { return (List<Usuario>) usuarioDao.findAll(); }
```

Figura 54- Implementación del servicio usuario (8)

Obtiene el JSON correspondiente a todos los usuarios registrados en la base de datos.

- *Update*:

```
@Override
public void put(Usuario usuario, long id_user) {
    usuarioDao.findById(id_user).ifPresent((x)->{
        usuario.setId_user(id_user);
        usuarioDao.save(usuario);
    });
}
```

Figura 55- Implementación del servicio usuario (9)

Este método actualiza el usuario con identificador "id_user" con los datos que están guardados en el atributo "Usuario".

- *Delete*:

```
@Override
public void delete(long id_user) { usuarioDao.deleteById(id_user); }
```

Figura 56- Implementación del servicio usuario (10)

Borra el usuario que posee el identificador "id_user".

Esto sería toda la implementación del paquete "Entity", una vez explicado esta parte pasamos a la implementación del paquete "Controller", en concreto, su clase "UsuarioController". Esta clase posee los siguiente identificadores y atributos:

```
@RestController
public class UsuarioController {

    @Autowired
    IUsuarioService usuarioService;
```

Figura 57- Implementación del servicio usuario (11)

El identificador @RestController posee una versión simplificada de @Controller que nos facilita la implementación de la API REST. Los métodos de esta clase son los siguientes:

```java
@GetMapping("/usuarios/{id_user}")
public Optional<Usuario> getOne(@PathVariable(value = "id_user") long id_user){
    return usuarioService.get(id_user);
}

@GetMapping("/usuarios")
public List<Usuario> getAllUsuarios() { return usuarioService.getAll(); }

@PostMapping(value = "/usuario")
public void add(@RequestBody Usuario usuario) { usuarioService.post(usuario); }

@PutMapping("/usuarios/{id_user}")
public void update(@RequestBody Usuario usuario, @PathVariable(value = "id_user") long id_user){
    usuarioService.put(usuario, id_user);
}

@DeleteMapping("/usuarios/{id_user}")
public void delete(@PathVariable(value = "id_user") long id_user) { usuarioService.delete(id_user); }
```

Figura 58- Implementación del servicio usuario (12)

Esta clase establece las rutas de cada método, haciendo referencia los de la clase de servicio. Los identificadores "*Mapping*" especifican esta ruta y la función principal de estos (CRUD).

Con esto ya tenemos la implementación del proyecto, aunque antes de lanzar la aplicación debemos modificar el archivo "aplication.properties", para establecer la ruta correcta del servidor y de la base de datos, además del usuario y contraseña, para poder acceder a esta.

```
spring.datasource.url=jdbc:mysql://localhost:3306/db_personas
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.platform=mysql
spring.jpa.hibernate.ddl-auto=create
spring.datasource.initialization-mode=always
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
```

Figura 59- Implementación del servicio usuario (13)

Accedemos a *localhost* y el puerto 3306, donde se encuentra el punto de entrada a la base de datos. El usuario y contraseña sería "root", y las demás propiedades son establecidas para que la conexión con la base de datos sea satisfactoria.

Por último, antes de empezar con la API REST de Python, cabe señalar que la ruta de conexión al *datasource* será el único cambio en el escalado a Docker, y Kubernetes.

PYTHON FLASK

Para la implementación del servicio "Python", se ha utilizado la herramienta Visual Code, que es un editor de texto.

El desarrollo de este servicio es sencillo, pero hay que conocer perfectamente los comandos de Python e instrucciones de la librería "Flask", además de tener instalado Python en nuestro ordenador, habiendo introducido las variables correspondientes en nuestro sistema. En este punto vamos a tratar la implementación de la primera versión

del servicio, ya que en la segunda versión solo se añade una ruta al archivo principal, y consideramos el cambio insignificante como para explicar las dos versiones.

Lo primero que debemos hacer es crear una carpeta donde va a estar alojado el proyecto, y desde donde vamos a trabajar. Una vez creada esta carpeta, abrimos un terminal de la consola en esa dirección local, e introducimos los siguientes comandos:

```
$ pip install virtualenv
$ virtualenv venv
```

El primer comando instala en nuestro sistema la librería “virtualenv” que utilizamos en el segundo comando para crear y configurar un entorno virtual en nuestro proyecto. Esto es necesario ya que debemos aislar la librerías y entornos de ejecución. Una vez creado el entorno virtual debemos activarlo, para ello debemos ejecutar los siguientes comandos

```
$ cd /venv/Scripts
$ actívate.bat
```

A partir de aquí deberíamos instalar en el entorno del proyecto las librerías que necesitamos en el proyecto, un ejemplo sería la librería “Flask”.

```
$ pip install flask
```

Considerando que ya hemos instalado todas las dependencias necesarias pasamos a la propia implementación del servicio, en este caso debemos crear una carpeta “src” que posea todo el código de nuestro proyecto. Añadimos las clases necesarias para el desarrollo del servicio.

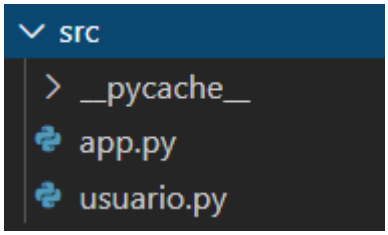


Figura 60- Implementación del servicio pyhton (1)

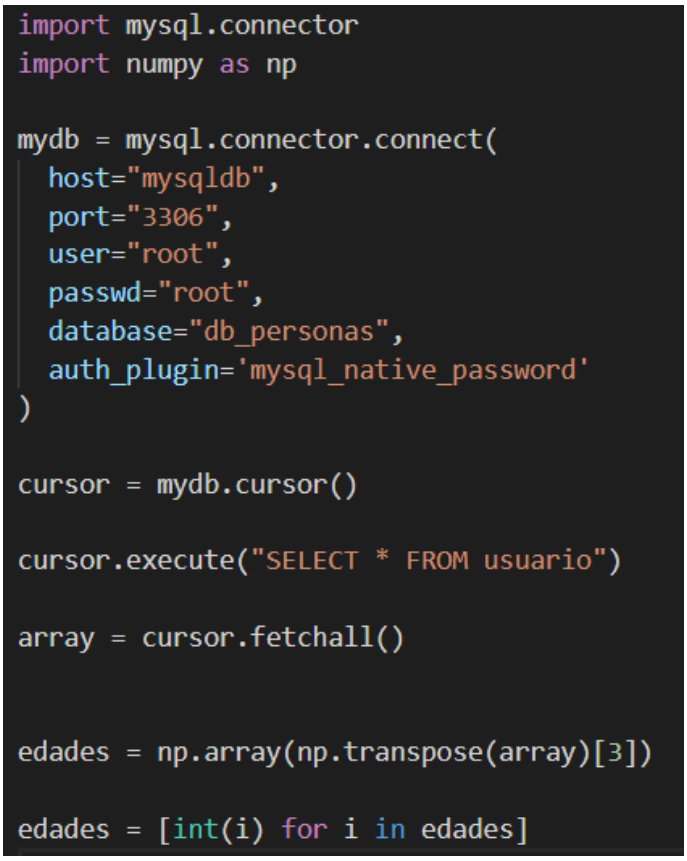
En nuestro caso, con estos archivos sería suficiente. El componente “app.py” funciona como controlador de la API REST definiendo las rutas de esta.

```python
from flask import Flask, jsonify
from usuario import edades

app = Flask(__name__)

@app.route('/', methods=['GET'])
def ping():
    return jsonify({"response": "hello world"})

@app.route('/edades', methods=['GET'])
def users2():
    return jsonify({"Las edades de los usuarios son": edades})

if __name__ == '__main__':
    app.run(host = "0.0.0.0", port="4000", debug=True)
```

Figura 61- Implementación del servicio pyhton (2)

Como podemos observar, necesitamos de dos paquetes de la librería “flask”, que nos ayudan en la implementación de la clase, y otro de la clase subyacente, “usuario”. Posee dos métodos GET que podemos utilizar, y la definición del host y puerto donde se va a exponer la API.

El componente “usuario.py” posee las instrucciones que posibilitan la interacción con la base de datos y el tratamiento de los datos que se van a exponer a través de la API.



```
import mysql.connector
import numpy as np

mydb = mysql.connector.connect(
  host="mysqldb",
  port="3306",
  user="root",
  passwd="root",
  database="db_personas",
  auth_plugin='mysql_native_password'
)

cursor = mydb.cursor()

cursor.execute("SELECT * FROM usuario")

array = cursor.fetchall()


edades = np.array(np.transpose(array)[3])

edades = [int(i) for i in edades]
```

Figura 62- Implementación del servicio pyhton (3)

En esta clase necesitamos la importación de la librería “mysql.connector” que realiza la conexión con la base de datos y “numpy” que es una librería estándar de Python que facilita el tratamiento de vectores y matrices (además de varias funciones derivadas del cálculo).

Lo primero que ejecuta es la definición de los parámetros necesarios para el acceso a los datos, posteriormente, ejecuta la consulta SQL, con la cual obtiene los datos relevantes de la tabla “usuario”. Accede a la columna de edades y las define como enteros, guardándolo en la variable “edades” que es la que importamos en el archivo “app.py”.

Por último, y con vistas a la implementación en Docker, ejecutamos el siguiente comando:

```
$ pip freeze > requirements.txt
```

Que guarda las dependencias de todo el proyecto en un archivo de texto, este archivo lo necesitaremos en la implementación del Dockerfile, con el objetivo de añadir al contenedor respectivo las dependencias pertinentes.

9.1.3. INTERFAZ GRÁFICA

Para la implementación del servicio “home”, se ha utilizado la herramienta PHP Storm, que es un IDE dedicado a proyectos web.

El desarrollo de este servicio es sencillo, ya que está formado por HTML5, CSS y Javascript.

La web se estructura en 4 apartados principales los cuales son:

- Index.php: Es la página de inicio de la web, desde aquí navegaremos a las distintas pantallas para probar los servicios.
- Empleados.php: Es la pantalla para probar el servicio de empleado en sus dos versiones.
- Python.php: Es la pantalla para probar el servicio de Python en sus dos versiones.
- Usuarios.php: Es la pantalla para probar el servicio usuario.

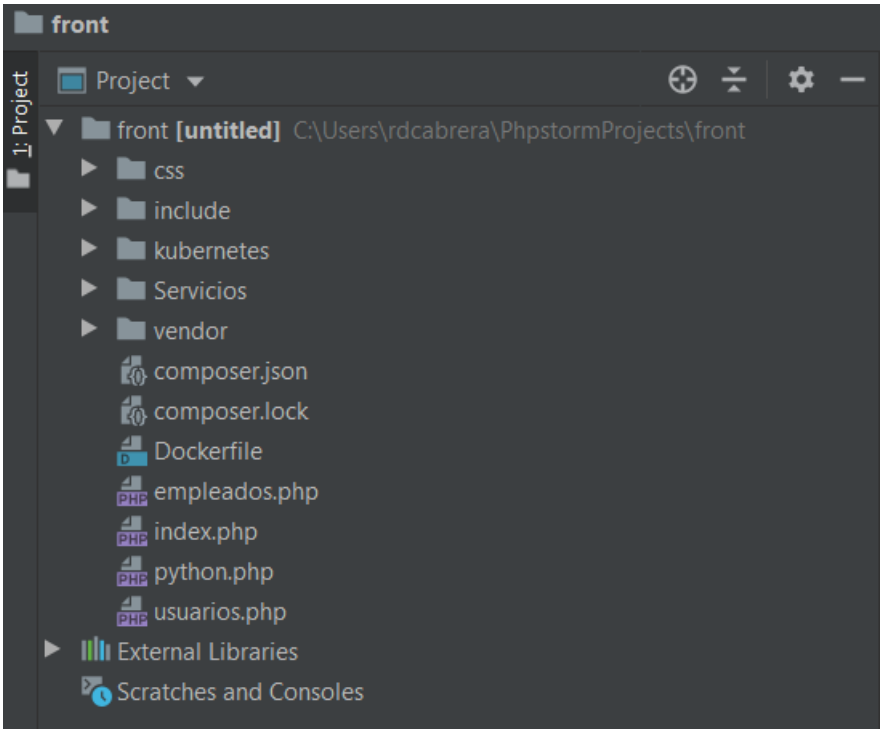


Figura 63- Estructura Web

Para llamar a los distintos servicios se usa GuzzelHTTP que es una librería de PHP que nos permite realizar peticiones de tipo REST, lo cual nos es muy útil, ya que todos los microservicios están implementados en REST y además podemos controlar los errores que nos lleguen.

```php
<?php
require("../../vendor/autoload.php");

$client = new GuzzleHttp\Client();
$uri = "http://empleado:8080/empleados";
$response = $client->request( method: 'GET', $uri,[
    'exceptions' => false, // Para que no muestre Excepciones
    'headers' => [
        'Accept'     => 'application/json'
    ]
]);
$response->getStatusCode(); // devuelve el Código de estado HTTP
$body = $response->getBody(); // Devuelve el contenido de la respuesta.

$array = json_decode($body, assoc: true);
//$data = json_decode( file_get_contents('http://localhost:8080/empleados'), true );

foreach ($array as $value) {
    echo "ID Empleado= " . $value['id_empleado'] . "\n";
    echo "ID Empresa= " . $value['empresa_id'] . "\n";
    echo "DNI= " . $value['dni_user'] . "\n";
    echo "Puesto= " . $value['puesto'] . "\n";
    echo "<br>";
}
?>

<?php require("../../include/pie.php"); ?>
</body>
</html>
```

Figura 64- Formulario del servicio Empleado

En la imagen anterior se observa cómo se llama al servicio empleado mediante la URI http://empleado:8080/empleados, la URI corresponde al servicio ubicado en el clúster de Kubernetes.

Para cada servicio se ha realizado un CRUD, así podemos probar los distintos verbos que nos ofrece REST.

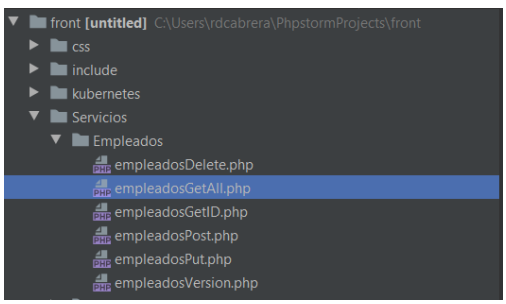


Figura 65- CRUD servicio empleado

Para la reutilización de código se crea un menú que se incluye mediante *<?php require("include/menu.php"); ?>*, así mismo el pie de página se incluye en un archivo aparte.

```
<!DOCTYPE html>
<head>
    <title>Home</title>
    <?php include "include/head-links.php"; ?>
</head>
<body>
<?php require("include/menu.php"); ?>

<?php require("include/pie.php"); ?>
</body>
</html>
```

Figura 66- index.php

Para los demás servicios se realiza el mismo procedimiento que en empleados, pero en distintos formularios para poder realizar las operaciones CRUD.

9.2. DESPLIEGUE CON DOCKER

En esta sección explicamos la implementación de nuestra aplicación en Docker, como expusimos en la sección de diseño, esta implementación se basa en la elaboración de un *Dockerfile* por cada microservicio y el despliegue de un contenedor por cada uno de estos.

9.2.1. INSTALACIÓN

El primer paso es ir a Docker Desktop[1], que es un apartado dentro de la página web Docker Hub. Nos debemos registrar para poder descargar el software para Windows 10, que es el sistema operativo de los d[1]os participantes del grupo.

La instalación de Docker en Windows 10 es muy simple, debido a que solo necesitamos hacer doble clic en el archivo ejecutable que hemos descargado. El asistente de instalación será el responsable de descargar todo lo que se necesita para que funcione correctamente.

Cuando termine el proceso de instalación, aparecerá una ventana que nos indica que se debe usar Hyper-V para que Docker pueda funcionar correctamente en Windows 10. Además, indica que si se tiene Virtual Box dejara de funcionar.

[1] Se puede acceder a Docker Desktop en el siguiente enlace: https://hub.docker.com/editions/community/docker-ce-desktop-windows

Después de activar esta opción, la computadora se reiniciará para aplicar los cambios sobre el sistema Operativo. Después de que se reinicie, se puede observar que Docker estará ejecutándose al momento que se inicia Windows.

Cuando finaliza la instalación, Docker se inicia automáticamente. El icono aparece en el área de notificación el cual indica que Docker se está ejecutando y es accesible desde una terminal. Para verificar que se instaló correctamente usamos un terminal de consola, en este caso de PowerShell.

Para verificar la versión de Docker que tenemos instalada ejecutamos el comando:

```
$ docker version
```

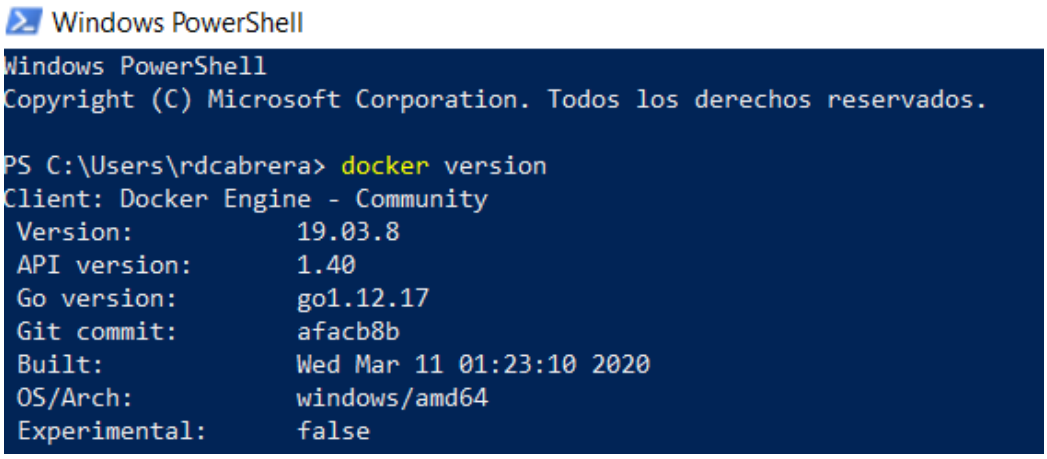


Figura 67- Comprobación de la instalación en Docker

9.2.2. ARQUITECTURA DEL SISTEMA

Una vez implementados los servicios, procedemos a virtualizarlos usando Docker. Para poder ejecutar los contenedores se necesitan imágenes base que permiten configurar y ejecutar los microservicios. En este caso como tenemos varios lenguajes en los distintos servicios (Python, PHP, Spring Boot, MySQL) es necesario partir de 4 imágenes.

Las imágenes se obtienen del registro de Docker Hub, donde elegimos las imágenes oficiales.

- *OpenJDK*: Para todos los servicios de las API REST, que están implementados en Spring Boot.
- Python: Para los dos servicios API REST que están implementados en Python.
- PHP: Para el servicio que implementa la funcionalidad de un interfaz.
- MySQL: Para el servicio de la base de datos.

Una vez elegidas las imágenes base, que contendrán los distintos contenedores. Se realizará el archivo de configuración denominado Dockerfile, donde se indica los comandos y configuración para poder desplegar los microservicios.

Al tener el Dockerfile nos permite crear una imagen de los microservicios, la cual es un sistema de capas a partir de distintos archivos. Vamos a introducir los Dockerfiles que hemos implementado.

```
FROM openjdk:8

COPY ./target/empleado-0.0.1-SNAPSHOT.jar empleado-0.0.1-
SNAPSHOT.jarWORKDIR /app

CMD ["java","-jar","empleado-0.0.1-SNAPSHOT.jar"]
```

Este es un ejemplo de un archivo Dockerfile que hemos implementado para los servicios desarrollados en Spring Boot, vamos a mostrar solo el ejemplo del servicio “Empleado”, ya que los demás siguen la misma plantilla. En este Dockerfile se han definido tres instrucciones:

- *FROM*: Establece la imagen base del contenedor, en este caso “openjdk:8”, siendo “openjdk” la imagen, y “8” la versión de esta. Cada vez que este comando aparezca en un Dockerfile, es para crear una nueva instancia de construcción de la imagen.
- *COPY*: Copia los archivos del origen, en nuestro caso el archivo Java que se encuentra en el directorio “target”, y la agrega a la ruta destino del contenedor, en este caso en el directorio “app”.
- *CMD*: Es la instrucción encargada de ejecutar el servicio dentro del contenedor.

Puesto que no queremos ser repetitivos, no vamos a mostrar la implementación de los demás Dockerfiles desarrollados para los microservicios en Spring Boot. El siguiente Dockerfile que nos muestra una configuración diferente es el del microservicio de Python.

```
FROM ubuntu:16.04

RUN apt-get update -y && \
    apt-get install -y python3-pip python3-dev && \
    pip3 install --upgrade pip

COPY ./requirements.txt /app/requirements.txt

WORKDIR /app

RUN pip3 install -r requirements.txt

COPY . /app

ENTRYPOINT [ "python3" ]

CMD [ "src/app.py" ]
```

Este Dockerfile posee nuevas instrucciones:

- *RUN*: Ejecutará el comando definido, en una nueva capa encima de la imagen y confirmará los resultados obtenidos. Como vemos esta instrucción se repite a lo largo del archivo, esto se debe a que las instrucciones deben seguir un orden adecuado, y como es el caso, podemos necesitar la repetición de ciertas instrucciones.
- *WORKDIR*: Establece un directorio de trabajo para el resto de los comandos, en este caso “app”.
- ENTRYPOINT: Esta instrucción funciona como un punto de entrada.

Debido a que el Dockerfile del microservicio en PHP se puede entender con lo explicado anteriormente, hemos decidido no mostrarlo en esta memoria.

A continuación, al tener ya definidos los Dockerfiles de los distintos microservicios se procede a crear las imágenes para poder desplegar los contenedores. Para poder crear las imágenes se van a ejecutar los comandos que vamos a mostrar en un terminal de consola.

COMANDOS PRINCIPALES DE DOCKER

En este apartado se exponen los comandos por pasos ascendentes que hemos seguido para la creación y despliegue individual de los contenedores.

```
$ docker build -t empleado-jdbc-v1 .
```

Nos permite construir la imagen, que adoptará el nombre de “empleado-jdbc-v1”, con ayuda del Dockerfile que existe en la ruta donde se ejecuta el comando.

```
$ docker network create empleados-mysql
```

Nos permite crear una red llamada “empleados-mysql”, para que los contenedores se puedan comunicar entre sí. Si ejecutamos el comando `$ docker inspect` sobre los contenedores desplegados con el *flag* “–network” y el nombre de esta red podremos comprobar que estos poseen una dirección de red que se ajusta al prefijo que posee la red creada.

```
$ docker run -name mysqldb –network empleados-mysql -e
MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=db_personas –mount
src=mysql-db-data, dst=/var/lib/mysql -d mysql:5.7
```

Este comando es el que ejecuta la imagen y nos crea el contenedor en local, en este caso se está creando el contenedor de la base de datos usando la imagen “mysql:5.7”, en la red “empleados-mysql”, con contraseña “root”, para el usuario “root” y base de datos “db_personas”, y con un volumen para que los datos sean persistentes.

Una vez que tenemos levantada la base de datos y creada nuestra red, se procede a crear los distintos contenedores relacionados con los microservicios.

```
$ docker container run --network empleados-mysql --name
usuario-jdbc-container -p 8081:8080 -d
ricarin/myrepository:usuarios-jdbc
```

Este es el ejemplo de un comando utilizado para desplegar el microservicio “Usuario”, que está construida en local y subida a un repositorio de Docker Hub.

El problema de ir ejecutando la aplicación de esta manera es que se vuelve tedioso al tener que repetir muchas veces los distintos comandos “docker run”. Para mejorar el despliegue de la aplicación se usa Docker Compose que nos permite crear varios scripts los cuales nos facilitan la construcción de los microservicios.

En el siguiente apartado, veremos el archivo docker-compose para crear el contendor de la base de datos y el microservicio de empleado.

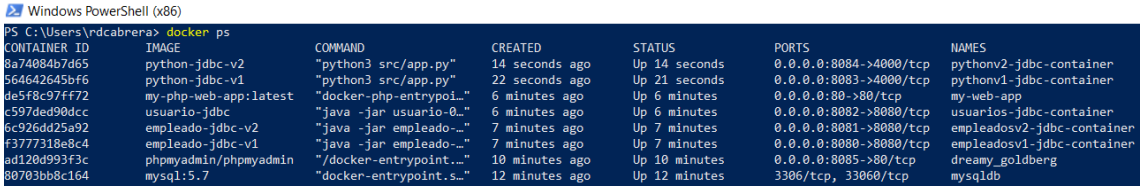
9.2.3. DESPLIEGUE DE LA APLICACIÓN

Para desplegar la aplicación se abre el terminal y se ejecuta el comando:

```
$ docker-compose up
```

```
version: "3"
services:
  employee-jdbc:
    image: employee-jdbc
    ports:
      - "8080:8080"
    networks:
      - employee-mysql
    depends_on:
      - mysqldb
  mysqldb:
    image: mysql
    networks:
      - employee-mysql
    environment:
      - MYSQL_ROOT_PASSWORD=root
- MYSQL_DATABASE=db_personas
    volumes:
      - "mysql-db-data:/var/lib/mysql"
    ports:
      - "33061:33060"
networks:
  employee-mysql:
volumes:
  mysql-db-data:
```

Una vez ejecutado los archivos docker-compose, se verifica en el terminal que están ejecutándose los contenedores utilizando el comando `$ docker ps`.



```
Windows PowerShell (x86)
PS C:\Users\rdcabrera> docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          PORTS                    NAMES
8a74084b7d65   python-jdbc-v2         "python3 src/app.py"     14 seconds ago   Up 14 seconds   0.0.0.0:8084->4000/tcp   pythonv2-jdbc-container
564642645bf6   python-jdbc-v1         "python3 src/app.py"     22 seconds ago   Up 21 seconds   0.0.0.0:8083->4000/tcp   pythonv1-jdbc-container
de5f8c97ff72   my-php-web-app:latest  "docker-php-entrypoi…"   6 minutes ago    Up 6 minutes    0.0.0.0:80->80/tcp       my-web-app
c597ded90dcc   usuario-jdbc           "java -jar usuario-0…"   6 minutes ago    Up 6 minutes    0.0.0.0:8082->8080/tcp   usuarios-jdbc-container
6c926dd25a92   empleado-jdbc-v2       "java -jar empleado-…"   7 minutes ago    Up 7 minutes    0.0.0.0:8081->8080/tcp   empleadosv2-jdbc-container
f3777318e8c4   empleado-jdbc-v1       "java -jar empleado-…"   7 minutes ago    Up 7 minutes    0.0.0.0:8080->8080/tcp   empleadosv1-jdbc-container
ad120d993f3c   phpmyadmin/phpmyadmin  "/docker-entrypoint.…"   10 minutes ago   Up 10 minutes   0.0.0.0:8085->80/tcp     dreamy_goldberg
80703bb8c164   mysql:5.7              "docker-entrypoint.s…"   12 minutes ago   Up 12 minutes   3306/tcp, 33060/tcp      mysqldb
```

Figura 68- Contenedores activos Docker por línea de comandos

Este comando nos ofrece un resumen del estado de nuestro ecosistema, proporcionándonos la siguiente información:

- *CONTAINER ID*: Identificador del contenedor.
- *IMAGE*: Imagen usada en ese contenedor.
- *COMMAND*: Comando ejecutado que como expusimos anteriormente se define con la instrucción "CMD" en el Dockerfile.
- *CREATED*: Tiempo transcurrido desde que se ejecutó.
- *STATUS*: Estado del contenedor.
- *PORTS*: Puerto donde se está exponiendo el contenedor.
- *NAME*: Nombre que sirve para identificar unívocamente, al igual que el identificador, pero útil para referirnos de forma rápida al contenedor.

Docker nos proporciona un *Dashboard* que nos permite ver los contenedores que se están ejecutando, así como sus logs, lo que nos es muy útil.

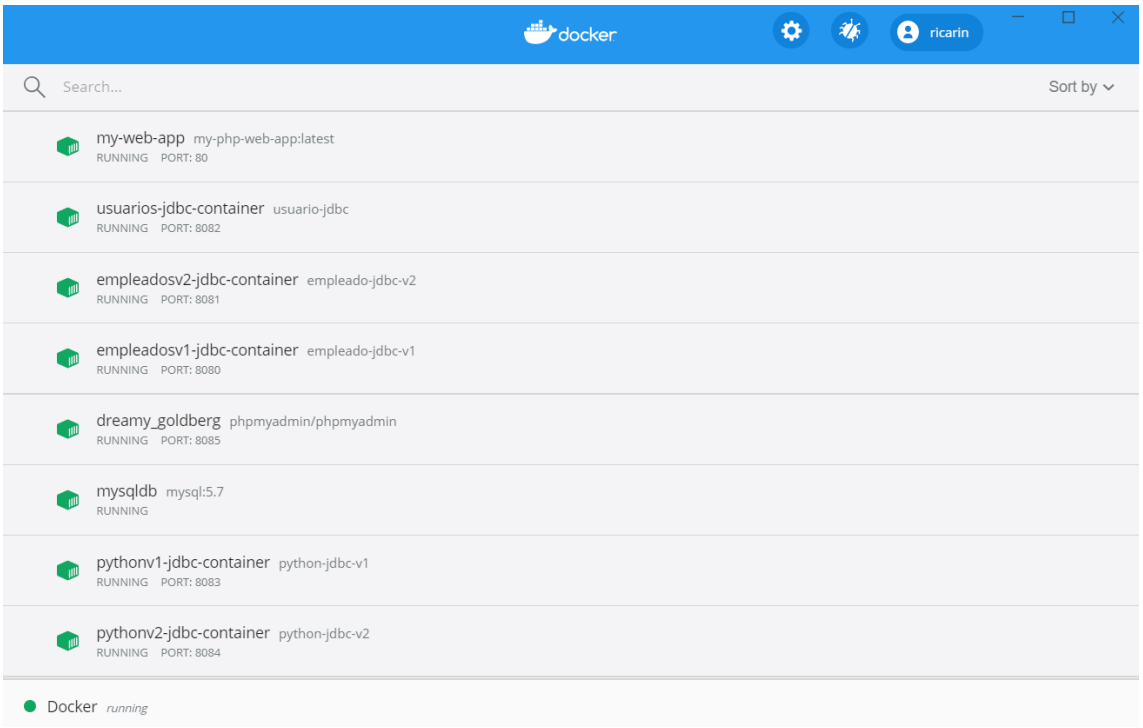


Figura 69- Contenedores activos Docker en panel

Si se quiere consultar el log solo se debe pinchar sobre el contenedor e ir al apartado correspondiente.

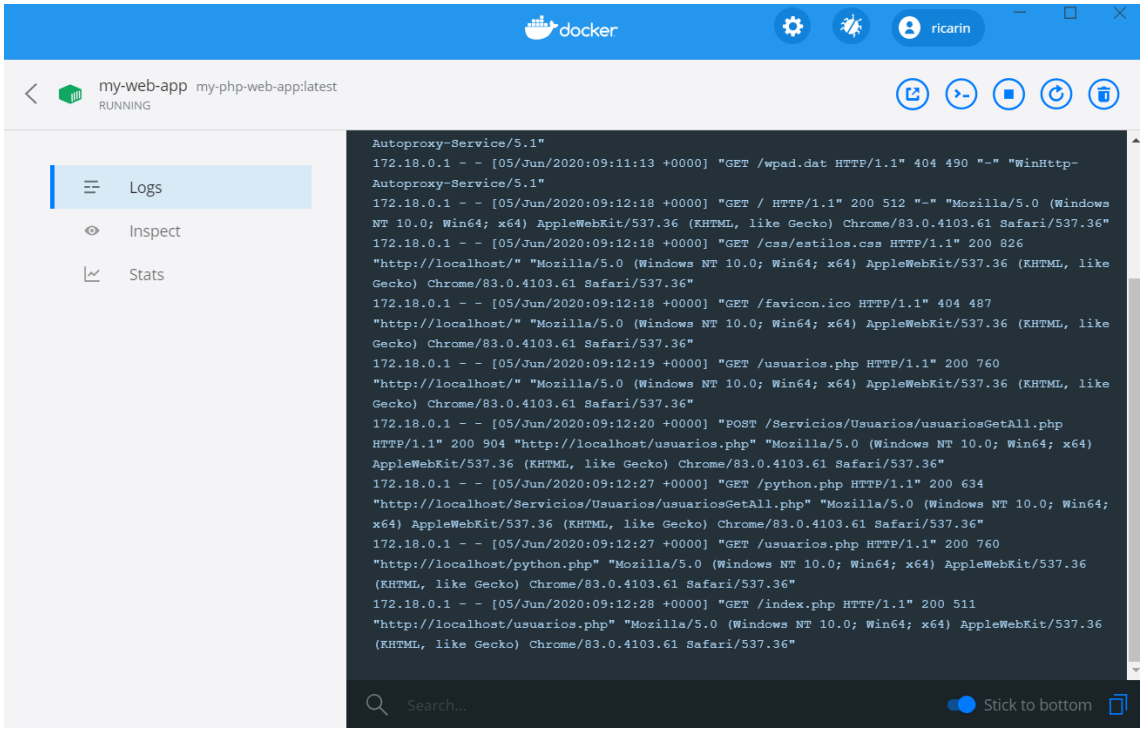


Figura 70- Prueba logs sobre contenedor Docker

Al tener los contenedores desplegados correctamente se incluirá Kubernetes para la orquestación. Pero antes, vamos a exponer algunas pruebas dentro de Docker, que pertenecen a la buena práctica en DevOps.

9.2.4. MEJORA DE DOCKER BUILD UTILIZANDO CACHE

En nuestro ordenador local, Docker tiene todas las capas anteriores de ejecución del comando "docker build" que ejecutamos. Es decir, cuando estamos trabajamos en ordenador local, la caché Docker se inicializará, por lo que recordará la ejecución de las instrucciones del comando anterior y optimizará la generación de imágenes, puesto que obtiene las capas de la caché y no las descarga.

Sin embargo, este no suele ser el caso en un entorno de integración continua, donde cada tarea de integración se realiza en una nueva computadora independiente creada para este propósito.

Docker ha resuelto este problema. Cuando ejecutamos un comando "docker build", podemos señalar que usa el caché de una imagen que se ha creado anteriormente. En otras palabras, si estoy construyendo una imagen *prueba-desa:latest*, se puede utilizar la caché, por ejemplo:

```
docker pull prueba-desa:latest

docker build -t prueba-desa:latest --cache- prueba-desa:latest .
```

9.2.5. TEST DE INTEGRACIÓN CON DOCKER

Tradicionalmente, probar aplicaciones es complejo, porque la ejecución de las distintas aplicaciones requiere tener la instalación con todas sus dependencias, y también puede requerir componentes adicionales. Sabemos que Docker resuelve estos problemas, y debido a que se utiliza la herramienta *docker-compose*, el proceso de ejecución local de la aplicación es realmente simple.

Las ventajas de usar la ejecución de prueba con Docker son las siguientes:

- **Ligero:** Docker es liviano, por lo que el archivo *docker-compose.yml* puede contener tantos servicios como se necesite y puede ejecutarse en una máquina. Esto permite realizar ejecuciones de distintos entornos complicados para pruebas las distintas pruebas.
- **Portable:** como el proceso se basa en Docker y este es portátil, se puede realizar pruebas en cualquier ordenador, sin importar su hardware interno o su sistema operativo.
- **Inmutable:** Dado que Docker es inmutable, cuando se crea una imagen y se verifica a través del proceso de integración continua se ejecutará de la misma manera en nuestro servidor de producción. Lo que permite evitar errores.

9.2.6. JIB EN CONTENEDORES JAVA

JIB es una herramienta que fue creada por Google, permite compilar los contenedores sin usar un dockerfile sin usar un demonio de Docker y sin un profundo dominio de las mejores prácticas de Docker. Está disponible como complementos para Maven y Gradle y como una biblioteca Java.

Jib permite organizar una aplicación en diferentes niveles, dependencias, recursos y clases. Jib utiliza el almacenamiento en caché de la capa de imagen de Docker como se explicó previamente, lo que permite garantizar que la compilación se pueda realizar rápidamente solo al volver a compilar los cambios.

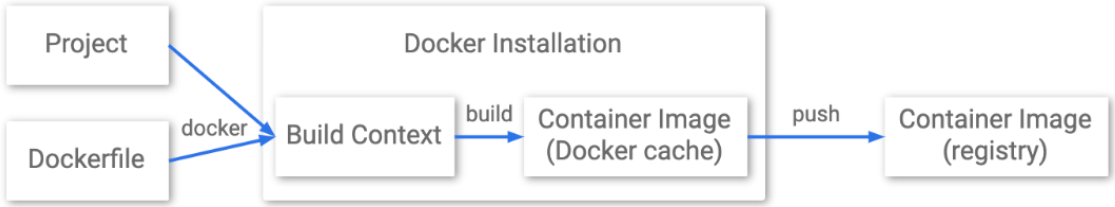


Figura 71- Flujo de compilación de Docker
Fuente: https://cloud.google.com/java/getting-started/jib

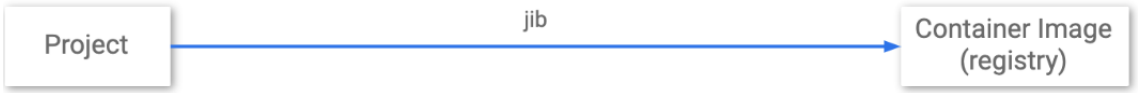


Figura 72- Flujo de compilación de Jib
Fuente: https://cloud.google.com/java/getting-started/jib

Como se puede observar en las imágenes es mucho más fácil y sencillo utilizar Jib lo que nos permite un despliegue rápido de los contenedores.

Las ventajas que proporciona son las siguientes:

- No es necesario usar el archivo Dockerfile para generar la imagen.
- Se integra con Gradel y Maven
- Optimiza el tiempo de desarrollo para generar la imagen de Docker.

9.2.7. EJEMPLO DE USO

Este es un ejemplo de código que ha sido añadido al archivo "ponm.xml" del Maven.

```
<plugin>
  <groupId>com.google.cloud.tools</groupId>
  <artifactId>jib-maven-plugin</artifactId>
  <version>2.0.0</version>
  <configuration>
    <to>
      <image>gcr.io/PROJECT/IMAGE</image>
    </to>
  </configuration>
</plugin>
```

Figura 73- Ejemplo de uso DevOps
Fuente: https://cloud.google.com/java/getting-started/jib

9.3. KUBERNETES

En esta sección explicamos la implementación de nuestra aplicación en Kubernetes, una vez que hemos terminado la implementación en Docker, es esencial escalar el proyecto a una gestión de la aplicación con un orquestador de contenedores.

9.3.1. INSTALACIÓN

Como ya adelantamos en el contexto, existen múltiples herramientas que nos ayudan con la ejecución de Kubernetes; Minikube, Minishift, Openshift, Kind, serían algunos ejemplos que nos presentan esta funcionalidad actualmente. En nuestro caso, los dos participantes del grupo hemos llegado a utilizar Minikube y la herramienta de Kubernetes que viene por defecto al instalarse Docker. La realización de la totalidad de la orquestación ha sido configurada utilizando estas dos herramientas, pero dado que, en la instalación de Istio hemos tenido dificultades con el acople de Minikube a la malla de servicios (todavía no se ha implementado correctamente Istio, para las especificaciones de nuestro sistema operativo, Windows), ya que hemos tenido colisiones de versiones con algunos archivos del ejecutable, decidimos continuar el proyecto utilizando solamente la opción por defecto de Docker.

Por tanto, en este apartado vamos a exponer la instalación de la opción por defecto de Docker, ya que es válida para la realización de toda la configuración y está disponible en cualquier sistema operativo.

El primer paso es abrir la aplicación que nos proporciona Docker, y seleccionar la opción de “Resources” en la barra lateral de la pantalla. Este menú nos posibilita cambiar la configuración de recursos que queremos utilizar para el proyecto, debido a que vamos a utilizar Kubernetes debemos aumentar los parámetros de estos recursos.

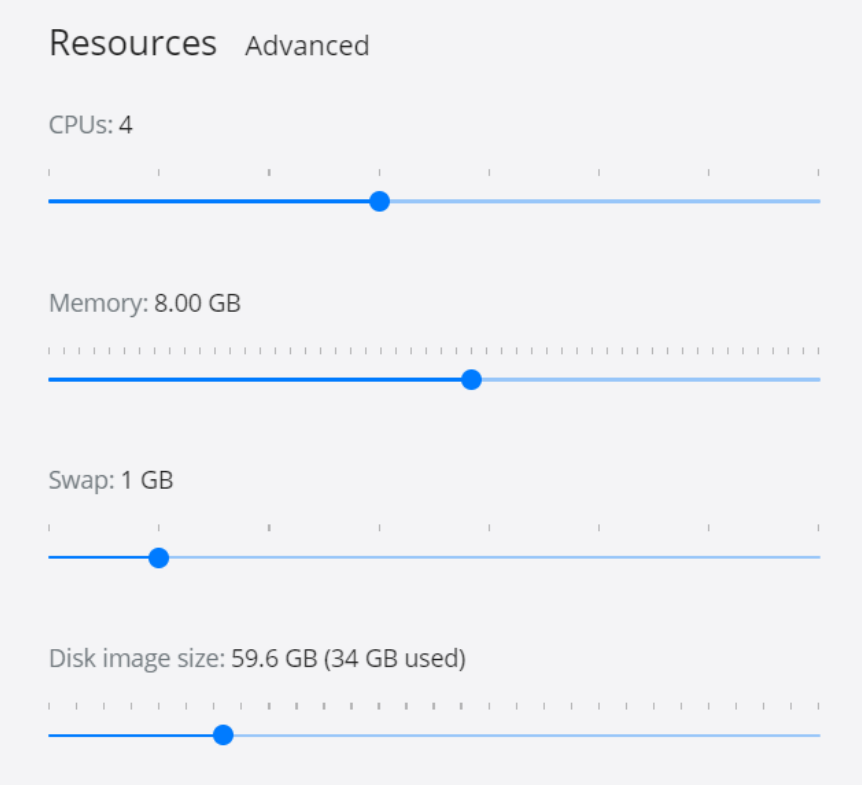


Figura 74- Recursos necesarios para la instalación de Kubernetes

En nuestro caso los recursos reservados son los mostrados en la figura anterior:

- 4 CPUs, en el caso de Minikube necesitaríamos alrededor de 4-6 CPUs.
- Memoria de 8GB, en Minikube dedicamos 16GB.
- Swap 1GB.
- Tamaño de la imagen del disco 59,6GB.

Una vez seleccionados los recursos pertinentes debemos pulsar el botón inferior izquierdo “Apply & Restart”, que guardará los cambios y reiniciará la aplicación de Docker.

El segundo paso es habilitar Kubernetes, para ello debemos seleccionar en el menú lateral la opción “Kubernetes”, y nos aparecerá una pantalla con tres aspectos importantes para el funcionamiento correcto.

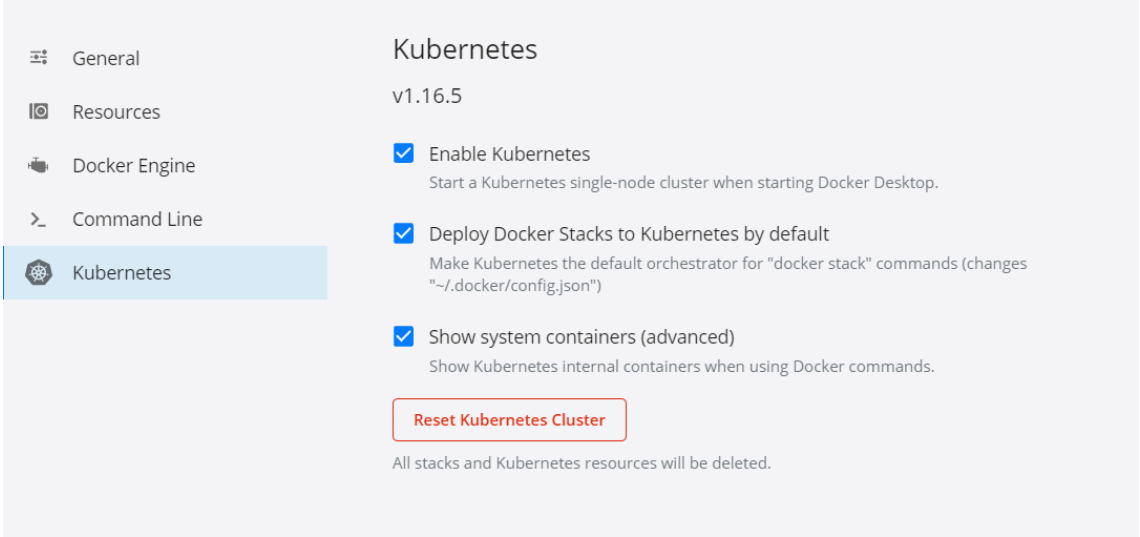


Figura 75- Parámetro que habilitar para la instalación de Kubernetes

Como podemos observar en la figura anterior debemos seleccionar las tres opciones, la primera habilita Kubernetes, la segunda proporciona a Kubernetes los permisos necesarios para situarlo como orquestador por defecto y la última da los permisos adecuados para que Kubernetes pueda observar el funcionamiento interno del sistema de Docker, para terminar, pulsamos sobre el botón "Start Kubernetes Cluster" y automáticamente después podremos usar el orquestador en nuestra consola.

Con el objetivo de comprobar la correcta instalación de Kubernetes, abrimos un terminal e introducimos el comando:

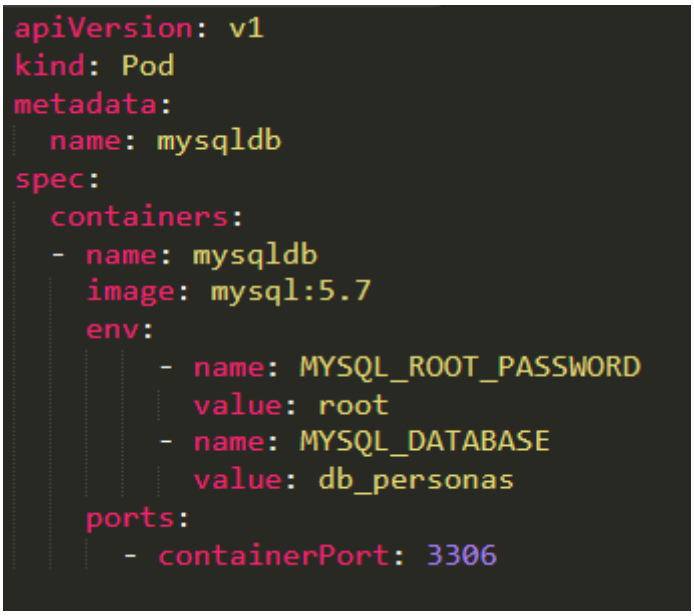
```
$ kubectl --help
```

Veremos en la salida el manual de parámetros de comando *kubectl*.

9.3.2. ARQUITECTURA DEL SISTEMA

Aunque el diseño final de la aplicación ya lo hemos expuesto, vamos a seguir los pasos que hicimos en el desarrollo de esta arquitectura. Con el objetivo de familiarizarnos con este orquestador de contenedores lo primero que hicimos fue listar los comandos más comunes de Kubernetes, dado que, en el caso de esta memoria los citaremos poco a poco vamos a comenzar directamente con la implementación.

La unidad más pequeña en Kubernetes es, como ya sabemos, un *pod*, por tanto, lo primero que desarrollamos fue el archivo de configuración del *pod* que albergaba el servicio de la base de datos.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysqldb
spec:
  containers:
  - name: mysqldb
    image: mysql:5.7
    env:
        - name: MYSQL_ROOT_PASSWORD
          value: root
        - name: MYSQL_DATABASE
          value: db_personas
    ports:
      - containerPort: 3306
```

Figura 76- Código implementación pod

La figura anterior muestra la configuración de un *pod* que soporta una base de datos MySQL, como podemos observar el formato es parecido a un formato JSON, donde tenemos pares clave-valor. En este tipo de archivos existen campos que son requeridos en cualquier configuración, estos son:

- *apiVersion*: Especifica la versión de la API de Kubernetes que queremos invocar. En este caso es la versión "v1", ya que los *pods* están soportados en esta.
- *kind*: Especifica el recurso que quieres crear, en este archivo, *pod*.
- *metadata:* Son los datos que identifican al objeto unívocamente. Hemos elegido "mysqldb" como nombre del objeto.
- *spec:* Este campo contiene la especificación de los contenedores.

Además de estos campos, existen otros dentro del propio campo "spec", que suelen ser muy variados según el tipo de recurso, en nuestro caso:

- *containers:* Campo que delimita la especificación de los contenedores utilizando subcampos.
- *name:* Nombre del contenedor que vamos a definir.
- *image:* Define la imagen que se va a desplegar en el contenedor.
- *env:* Define el ecosistema del contenedor, dentro de este campo se encuentran:
    - *name:* Nombre del recurso del ecosistema que queremos especificar. En nuestro caso se define la contraseña de la base de datos y el nombre de esta.
    - *value:* Valor del componente antes especificado.
- *ports:* Definición de los puertos donde se va a exponer el *pod*.
- *containerport:* Número del puerto.

Con el archivo de configuración ya implementado, debemos ejecutar el siguiente comando para lanzar el *pod*:

```
$ kubectl apply -f <nombre de archivo>
```

Si el archivo se mantiene en el marco de sintaxis que Kubernetes tiene definido deberemos obtener la siguiente salida.

```
C:\Users\rauld\Documents\GitHub\TFG\Proyecto-TFG\kubernetes\pod>kubectl apply -f pod-bbdd.yaml
pod/mysqldb created
```

Figura 77- Creación del pod

Por último, para comprobar que se ha ejecutado correctamente y que el contenedor está corriendo, debemos ejecutar:

```
$ kubectl get pods
```

Y deberemos obtener la siguiente salida:

```
C:\Users\rauld\Documents\GitHub\TFG\Proyecto-TFG\kubernetes\pod>kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
mysqldb   1/1     Running   0          27s
```

Figura 78- Consulta del pod

Como podemos observar el terminal muestra varios campos, estos definen las siguientes características:

- NAME: Nombre del recurso, en este caso es un objeto *pod*.
- READY: Contiene una fracción donde el numerador muestra el número de contenedores que se están ejecutando, y el denominador muestra el número de contenedores que se definieron en el archivo de configuración.
- STATUS: Estado del *pod*.
- RESTARTS: Número de reinicios que ha tenido que hace Kubernetes, normalmente debido a fallos en el funcionamiento del contenedor o interacción con este.
- AGE: Tiempo que lleva desplegado el objeto.

Estos serían los pasos que debemos seguir cuando desplegamos cualquier recurso, para comprobar que todo funcione correctamente. Dado que es un contenedor corriendo MySQL debemos introducir los datos pertinentes, esto se hace ejecutando el siguiente comando:

```
$ kubectl exec -it <nombre del pod> -- mysql -u root -p
```

El comando *exec* con los *flags* "-it" nos permiten situarnos dentro del contenedor (si tenemos más de un contenedor en el pod de veremos especificar el nombre del contenedor siguiendo el patrón "-c <nombre del contenedor>") ejecutando la consola interactiva del sistema operativo en cuestión. Además, Kubernetes nos obliga a introducir los comandos que deseamos ejecutar, introduciendo dos guiones y el comando con un espacio de separación, en nuestro caso abrimos la consola de MySQL para interactuar directamente con las bases de datos (normalmente el comando *exec* se introduce con la opción "-- /bin/bash" que ejecuta un terminal). El parámetro "-p" indica la contraseña de la base de datos, que introduciríamos una vez ejecutado el comando. La salida del terminal deberá ser algo parecido a:

```
C:\Windows\system32>cd C:\Users\rauld\Documents\GitHub\TFG\Proyecto-TFG\kubernetes\pod

C:\Users\rauld\Documents\GitHub\TFG\Proyecto-TFG\kubernetes\pod>kubectl exec -it mysqldb -- mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.7.29 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use db_personas;
Database changed
mysql>
```

Figura 79- Ejecución de la consola interactiva del contenedor MySQL

Una vez que estamos dentro del terminal seleccionaríamos la base de datos que definimos en el campo "env" e introduciríamos las instrucciones de los archivos SQL mostrados en la sección anterior.

En vista de que todo funciona correctamente, pasamos a implementar todos los archivos de los *pods* que necesitamos. Dado que, son archivos parecidos no lo vamos a mostrar.

El siguiente paso es implementar los *deployments* y *services* de cada servicio para poder desplegar toda la aplicación, pero antes de mostrar estos archivos vamos a mostrar los archivos de los volúmenes y secretos necesarios. En nuestro caso son tres archivos en total:

- *Persistent Volume* y *Persistent Volume Claim*

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: volumen
  labels:
    type: local
spec:
  volumeMode: Filesystem
  storageClassName: mysql
  persistentVolumeReclaimPolicy: Retain
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteOnce
  hostPath:
      path: "/media/data"
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  volumeMode: Filesystem
  storageClassName: mysql
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Figura 80- Implementación Volumes Kubernetes

Definimos un volumen de 10GB de capacidad, con política de solo lectura y en la ruta "/media/data", de esta manera los datos guardados en la base de datos serán persistentes.

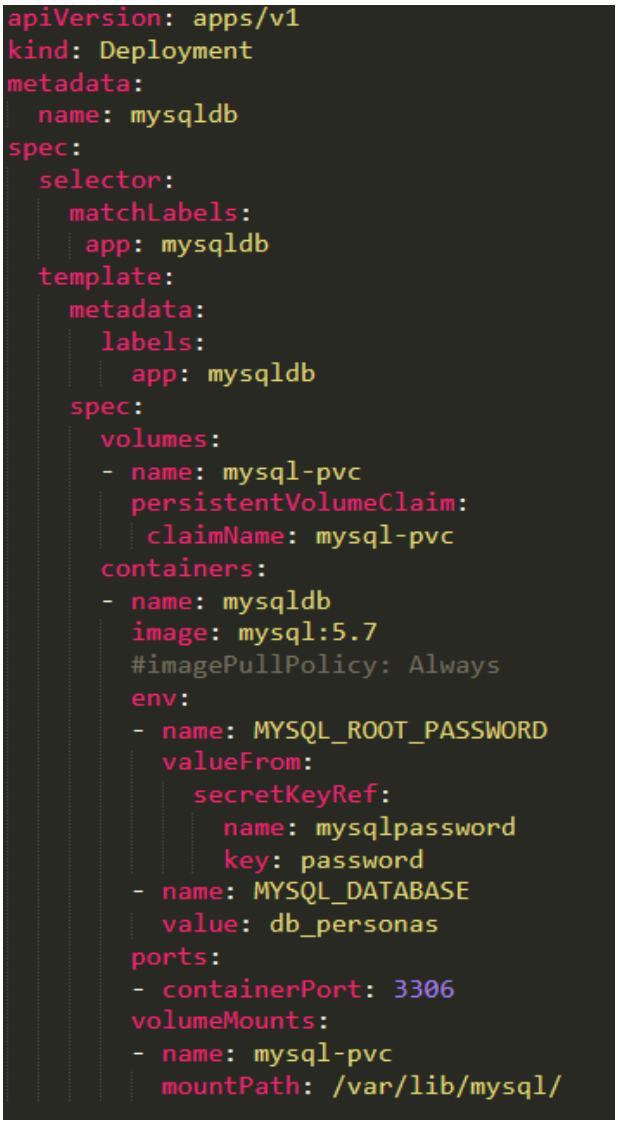
- *Secret*.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysqlpassword
type: Opaque
data:
  password: root
```

Figura 81- Implementación Secret Kubernetes

Definimos un secreto que contiene la contraseña de la base de datos y es de tipo opaco, de esta manera no se escribirá en texto plano en el *deployment* de la base de datos.

Pasamos a implementar los *deployments* de cada servicio, en este caso vamos a mostrar el de la base de datos y el del servicio usuario, ya que los demás son muy similares entre sí. El *deployment* de la base de datos es el siguiente:



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysqldb
spec:
  selector:
    matchLabels:
     app: mysqldb
  template:
    metadata:
      labels:
        app: mysqldb
    spec:
      volumes:
      - name: mysql-pvc
        persistentVolumeClaim:
         claimName: mysql-pvc
      containers:
      - name: mysqldb
        image: mysql:5.7
        #imagePullPolicy: Always
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysqlpassword
              key: password
        - name: MYSQL_DATABASE
          value: db_personas
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-pvc
          mountPath: /var/lib/mysql/
```

Figura 82- Implementación Deployment Base de datos Kubernetes

En este archivo seleccionamos el objeto “Deployment”, y especificamos el campo *spec*. Puesto que ya hemos expuesto la configuración de un *pod* vamos a explicar los campos que son nuevos:

- *volumes:* Especifica el volumen que va a utilizar la base de datos.
- *selector y template:* Este campo define los *labels* necesarios para identificar y seleccionar el *deployment* inequívocamente, esto nos servirá para poder desarrollar los *services*.
- *env:* En este caso lo único diferente es la introducción del secreto en vez de la contraseña en texto plano.
- *volumeMounts:* También dedicado al uso de los volúmenes, pero esta vez determinamos la ruta dentro del contenedor donde se va a guardar la información.

El *deployment* del servicio usuario es el siguiente:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: usuario-jdbc
spec:
  selector:
    matchLabels:
     app: usuario-jdbc
  template:
    metadata:
      labels:
        app: usuario-jdbc
    spec:
      containers:
      - name: usuario-jdbc
        image: rauldieg/myrepository:usuario-jdbc
        imagePullPolicy: Always
        ports:
        - containerPort: 8080
      imagePullSecrets:
      - name: myregistrykey
```

Figura 83- Implementación Deployment Usuario Kubernetes

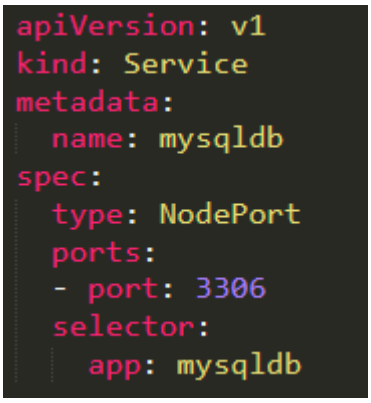
De este *deployment* lo único que no hemos abordado es la utilización de una imagen privada, es decir, el uso de una imagen creada con Docker. Debemos registrar un secreto nuevo con nuestras credenciales de Docker Hub, crear un repositorio y subir la imagen en cuestión a este repositorio, para posteriormente como se ve en la figura introducir este secreto en el *deployment*, la política de descarga la hemos definido como "Always", ya que queremos que descargue automáticamente la imagen de nuestro repositorio teniendo en cuenta que puede haber sido actualizada. Los comandos que desarrollan los pasos anteriores son los siguientes:

```
$ kubectl create secret <nombre de repositorio> --docker-
server=docker.io --docker-username=<usuario de Docker hub> --
docker-password=<contraseña> --docker-email=<email>
```

```
$ docker tag <nombre de la imagen>
<usuario>/<repositorio>:<nombre de la imagen>
```

```
$ docker push <usuario>/<repositorio>:<nombre de la imagen>
```

En este momento podremos lanzar los servicios mostrados, pero no se comunicarán entre ellos, hasta que desarrollemos y despleguemos los sendos *services*. En este caso vamos a mostrar solo uno, debido a que son similares.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysqldb
spec:
  type: NodePort
  ports:
  - port: 3306
  selector:
    app: mysqldb
```

Figura 84- Implementación Service Kubernetes

Como podemos observar definimos un *service* de tipo *NodePort* y que expondrá el puerto 3306 (es el puerto que definimos en el *deployment* de la base de datos) en un puerto local que estará en el rango 30000-32767 (lo elige el propio orquestador), y por último busca la *app* "mysqldb" como referencia.

Desplegamos los services y probamos que se conectan correctamente con Postman. El comando que debemos ejecutar para cerciorarnos de que todo funciona correctamente es:

```
$ kubectl get all
```

Que mostrará una salida parecida a:

```
C:\Users\rauld\Documents\GitHub\TFG\Proyecto-TFG\kubernetes>kubectl get all
NAME                                 READY   STATUS    RESTARTS   AGE
pod/mysqldb-764f6fb4db-5qcgm         1/1     Running   0          55m
pod/usuario-jdbc-85477cd9f4-mm5vx    1/1     Running   0          9s

NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP          4d20h
service/mysqldb         NodePort    10.104.246.180   <none>        3306:30537/TCP   17s
service/usuario-jdbc    NodePort    10.98.94.47      <none>        8080:32293/TCP   3s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mysqldb        1/1     1            1           65m
deployment.apps/usuario-jdbc   1/1     1            1           9s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/mysqldb-764f6fb4db        1         1         1       55m
replicaset.apps/usuario-jdbc-85477cd9f4   1         1         1       9s
```

Figura 85- Consulta de los pods, services, deployments y replica sets

En esta figura nos muestra varios objetos desplegados, los primeros son los pods, que ya expusimos, los segundos son los *services*, que poseen los siguientes aspectos:

- En este caso tenemos tres *services* desplegados, ya que Kubernetes nos proporciona un servicio de tipo *ClusterIP*, del clúster en general. Los otros dos son los *services* del servicio "Usuario" y la bbdd.
- Podemos conocer el tipo de *service*, la dirección del clúster y externa, el puerto escogido, y la edad del recurso.

El tercer recurso son los *deployments*, el campo UP-TO-DATE nos informa el número de replicas que se han actualizado y el campo AVAILABLE muestra el número de replicas

disponibles. El cuarto recurso son las *replica sets*, que nos muestra el número deseado de replicas y el actual. Estas *replica sets* como ya avanzamos en el contexto y en el diseño son definidos por los *deployments*.

Una vez explicado esto, vamos a mostrar una captura de la prueba de esta parte en Postman.

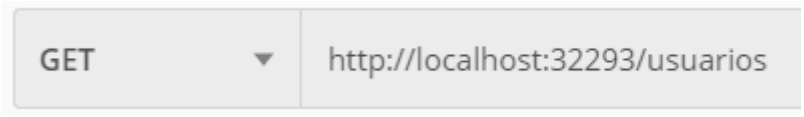


Figura 86- Consulta con Postman

Enviamos una petición GET al servicio usuario, con la ruta específica para obtener el JSON con todos los datos de la tabla "usuario". En este caso como podemos observar el puerto elegido es el que obtenemos del terminal, como vimos anteriormente. El resultado de esta petición es:


```
[
    {
        "id_user": 1,
        "name": "Raul",
        "edad": 21,
        "dni": 5784965
    },
    {
        "id_user": 2,
        "name": "Pedro",
        "edad": 45,
        "dni": 5796548
    },
    {
        "id_user": 3,
        "name": "Luis",
        "edad": 12,
        "dni": 68593741
    },
```


Figura 87- Resultado de la consulta en Postman

Con esta prueba podemos afirmar que se comunican correctamente. Antes de continuar con el desarrollo de los demás servicios, vamos a exponer la opción de utilizar un clúster virtual donde desplegar nuestra aplicación, ya que en este momento lo estamos desplegando todo en el *namespace* por defecto. En este caso lo único que debemos añadir a los comandos de creación de objetos es la opción "--namespace=<nombre del namespace>", esto debe ser añadido, una vez que hemos creado el respectivo *namespace*, con el comando:

```
$ kubectl create ns <nombre del namespace>
```

De esta manera si ejecutamos el comando:

```
$ kubectl get ns
```

Podremos comprobar que posemos un nuevo espacio de nombres.

Por último, antes de pasar al apartado del despliegue de toda la aplicación debemos hablar sobre la implementación de los *deployments*, ya que debemos desarrollar un *deployment* con el mismo valor del campo *app* para las distintas versiones, pero sin que

se remplace por las anterior. Para esto Kubernetes nos ofrece una opción particular para estos casos:

```yaml
metadata:
  labels:
    app: empleado
    version: v2
```

```yaml
selector:
  matchLabels:
    app: empleado
    version: v2
```

```yaml
template:
  metadata:
    labels:
      app: empleado
      version: v2
```

Figura 88- Implementación versiones Kubernetes

Como podemos observar, debemos definir una versión "X" en los campos *metadata*, *selector* y *template*. De esta manera podemos estructurar nuestra aplicación para que el *service* apunte a todas las versiones de esa *app*.

9.3.3. DESPLIEGUE DE LA APLICACIÓN

Para este caso, podemos situar todos los services y deployments de la aplicación en una misma carpeta, así podremos hacer uso del comando general que se encarga de desplegar todos los objetos en una misma ruta:

```
$ kubectl apply -f .
```

El resultado de levantar toda la aplicación es el siguiente:

```
C:\Users\rauld\Documents\GitHub\TFG\Proyecto-TFG\kubernetes>kubectl get all
NAME                                      READY   STATUS    RESTARTS   AGE
pod/empleado-jdbc-v1-548669f5f8-rjwfc     1/1     Running   0          79s
pod/empleado-jdbc-v2-74dfd87b5b-zjjsw     1/1     Running   0          77s
pod/my-php-web-app-bdc78d574-sx9f5        1/1     Running   0          9s
pod/mysqldb-764f6fb4db-5qcgm              1/1     Running   0          114m
pod/python-jdbc-v1-558f8d8d8f-ntwb4       1/1     Running   0          19s
pod/python-jdbc-v2-6fd7c4d46c-9zmr6       1/1     Running   0          23s
pod/usuario-jdbc-85477cd9f4-mm5vx         1/1     Running   0          59m

NAME                     TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/empleado         NodePort    10.107.31.107    <none>        8080:30877/TCP   2m41s
service/kubernetes       ClusterIP   10.96.0.1        <none>        443/TCP          4d21h
service/my-php-web-app   NodePort    10.102.102.143   <none>        80:32398/TCP     12s
service/mysqldb          NodePort    10.104.246.180   <none>        3306:30537/TCP   59m
service/python           NodePort    10.100.0.239     <none>        4000:31974/TCP   28s
service/usuario-jdbc     NodePort    10.98.94.47      <none>        8080:32293/TCP   59m

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/empleado-jdbc-v1     1/1     1            1           79s
deployment.apps/empleado-jdbc-v2     1/1     1            1           77s
deployment.apps/my-php-web-app       1/1     1            1           9s
deployment.apps/mysqldb              1/1     1            1           124m
deployment.apps/python-jdbc-v1       1/1     1            1           19s
deployment.apps/python-jdbc-v2       1/1     1            1           23s
deployment.apps/usuario-jdbc         1/1     1            1           59m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/empleado-jdbc-v1-548669f5f8     1         1         1       79s
replicaset.apps/empleado-jdbc-v2-74dfd87b5b     1         1         1       77s
replicaset.apps/my-php-web-app-bdc78d574        1         1         1       9s
replicaset.apps/mysqldb-764f6fb4db              1         1         1       114m
replicaset.apps/python-jdbc-v1-558f8d8d8f       1         1         1       19s
replicaset.apps/python-jdbc-v2-6fd7c4d46c       1         1         1       23s
replicaset.apps/usuario-jdbc-85477cd9f4         1         1         1       59m
```

Figura 89- Despliegue de la aplicación en Kubernetes

Como podemos observar, la aplicación, en resumen, está formada por siete *pods* (uno para cada servicio) y 5 *services* (uno por cada servicio, menos las dos versiones). Si, además, queremos comprobar que la aplicación está configurada correctamente, podemos introducir en un navegador la ruta y puerto de entrada a la aplicación y comprobar las funcionalidades, como ejemplo mostramos la de la página "Usuario".

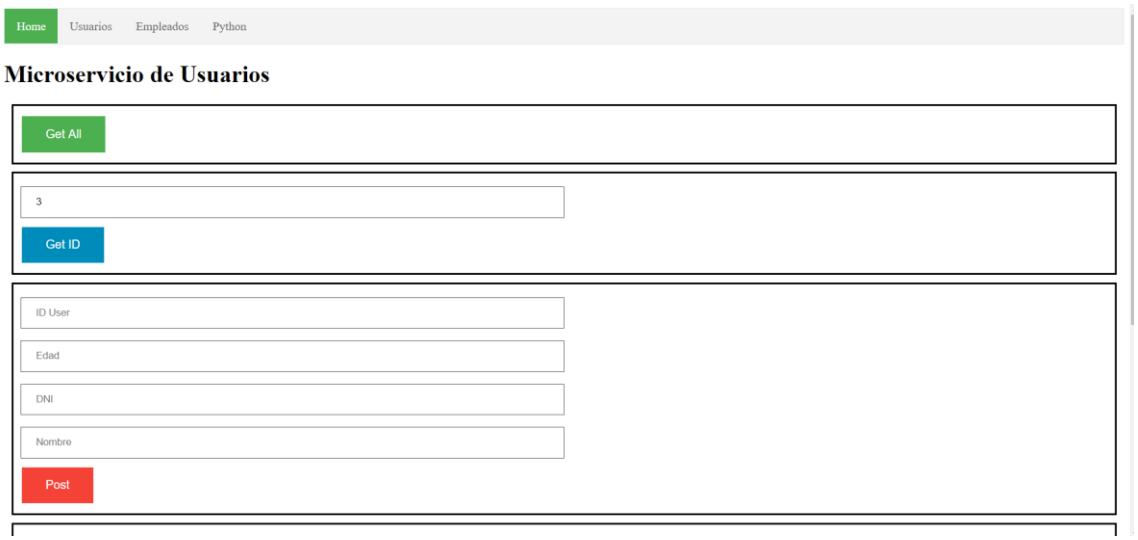


Figura 90- Pantalla del servicio Usuario

Salida de la petición mostrada en la figura anterior:

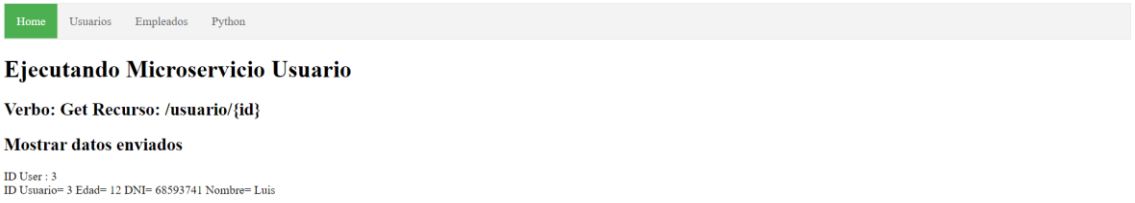


Figura 91- Petición GET sobre el servicio Usuario

9.4. INTEGRACIÓN CON ISTIO

Seguimos con la explicación de la integración con Istio. Es el último paso para obtener nuestra malla de servicios. Partimos de la configuración que hemos mostrado en Kubernetes, y vamos a ir escalando la aplicación utilizando Istio; inyección del patrón sidecar, control del tráfico, seguridad, observabilidad, etc.

Siguiendo la estructura de las secciones anteriores, comenzamos con el apartado de instalación de esta plataforma.

9.4.1. INSTALACIÓN

Lo primero que hicimos fue descargar la versión adecuada de Istio. Dentro de las versiones que estaban disponibles, decimos escoger la versión 1.5.1.

Una vez que ya disponemos del directorio de Istio, accedemos con un terminal a este, e introducimos el siguiente comando:

```
$ istioctl manifest apply --set profile=demo
```

Este comando aplica los cambios del archivo “manifest.yml” sobre el clúster principal, instalando un nuevo espacio de nombre “istio-system”, la siguiente figura muestra que se ha añadido correctamente.

```
C:\istio-1.5.1>kubectl get ns
NAME              STATUS   AGE
default           Active   8d
docker            Active   8d
istio-system      Active   35s
kube-node-lease   Active   8d
kube-public       Active   8d
kube-system       Active   8d
tfg               Active   2m28s
```

Figura 92- Consulta de namespaces activos

Como vemos el nuevo espacio de nombre se ha creado correctamente. El siguiente paso es comprobar que los *services* están todos funcionando.

```
C:\istio-1.5.1>kubectl get svc -n istio-system
NAME                     TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)
                                        AGE
grafana                  ClusterIP      10.108.9.174     <none>        3000/TCP
                                        3m24s
istio-egressgateway      ClusterIP      10.109.105.105   <none>        80/TCP,443/TCP,15443/TCP
                                        3m26s
istio-ingressgateway     LoadBalancer   10.97.194.163    localhost     15020:31784/TCP,80:30470/TCP,44
2:30445/TCP,31400:30768/TCP,15443:31509/TCP   3m25s
istio-pilot              ClusterIP      10.101.15.163    <none>        15010/TCP,15011/TCP,15012/TCP,8
                                        3m36s
istiod                   ClusterIP      10.111.22.61     <none>        15012/TCP,443/TCP
                                        3m36s
jaeger-agent             ClusterIP      None             <none>        5775/UDP,6831/UDP,6832/UDP
                                        3m24s
jaeger-collector         ClusterIP      10.109.78.175    <none>        14267/TCP,14268/TCP,14250/TCP
                                        3m24s
jaeger-collector-headless   ClusterIP   None             <none>        14250/TCP
                                        3m24s
jaeger-query             ClusterIP      10.102.181.86    <none>        16686/TCP
                                        3m24s
kiali                    ClusterIP      10.96.48.43      <none>        20001/TCP
                                        3m24s
prometheus               ClusterIP      10.98.153.147    <none>        9090/TCP
                                        3m24s
tracing                  ClusterIP      10.111.75.133    <none>        80/TCP
                                        3m24s
zipkin                   ClusterIP      10.104.208.165   <none>        9411/TCP
                                        3m24s
```

Figura 93- Consulta de los services activos en el namespace “istio-system”

Si observamos los nombres de los *services* activos en el espacio de nombres de Istio, nos damos cuenta de que la mayor parte de recursos o funcionalidades que nos ofrece Istio y que hemos explicado en el contexto, poseen un *service* y como veremos más a continuación un *pod*, esto se debe a que las propias funcionalidades son manejadas en un *pod* propio. Comprobamos que los *pods* también están funcionando correctamente.

```
C:\istio-1.5.1>kubectl get pods -n istio-system
NAME                                    READY   STATUS    RESTARTS   AGE
grafana-5cc7f86765-9rdn6                1/1     Running   0          8m3s
istio-egressgateway-598d7ffc49-n86cf    1/1     Running   0          8m4s
istio-ingressgateway-7bd5586b79-jb7bl   1/1     Running   0          8m4s
istio-tracing-8584b4d7f9-v26ns          1/1     Running   0          8m3s
istiod-646b6fcc6-xhk4t                  1/1     Running   0          8m14s
kiali-696bb665-7jpsg                    1/1     Running   0          8m2s
prometheus-6c88c4cb8-5wjr2              2/2     Running   0          8m2s
```

Figura 94- Consulta de los pods activos en el namespace "istio-system"

El último paso de la instalación sería ejecutar un ejemplo básico, como puede ser un *pod*. Dado que en la propia web de Istio existen varios archivos implementados, vamos a pasar a la siguiente explicación, ya que no creemos innecesario mostrar este último paso.

9.4.2. INYECCIÓN DEL PATRON SIDECAR

Istio nos ofrece dos maneras eficientes de inyectar en nuestros *pods* el *proxy* Envoy. Dependiendo de las características de nuestra aplicación deberemos escoger una u otra.

INYECCIÓN MANUAL

Esta opción nos permite elegir manualmente que *pods* deberán tener la configuración del proxy Envoy. El siguiente comando escoge un archivo de formato YMAL que define un *pod,* y crea otro con la configuración del contenedor Envoy.

```
$ istioctl kube-inject -f <nombre del archivo original> -o
<nombre del archivo nuevo>
```

A partir de este momento si aplicamos este archivo nuevo veremos que se inicia con un contenedor más que el archivo original, esto nos indicará que el proxy Envoy ha sido introducido correctamente.

Las ventajas de esta manera de inyección es que nos permite discernir entre los archivos que van a formar la malla de servicios y los que permanecerán iguales. La gran desventaja es que deberemos introducir el comando anterior un número de veces igual al número de *pods* que tengamos en nuestra aplicación, y como es una arquitectura de microservicios normalmente son bastantes archivos.

INYECCIÓN AUTOMÁTICA

Istio nos permite configurar por defecto la inyección del proxy Envoy ejecutando el siguiente comando que etiqueta a un espacio de nombre con "istio-injection=enabled".

```
$ kubectl label ns <namespace> istio-injection=enabled
```

De esta manera a cualquier *pod* que se despliegue en ese espacio de nombres se le inyectará el proxy Envoy. Como podemos observar esta manera es mucho más rápida e intuitiva que la anterior.

Además, de como ya hemos comentado el *pod* va a tener un contenedor más, hay más formas de asegurarnos de que el *proxy* Envoy esté funcionando correctamente. Istio nos ofrece una guía (El enlace es el siguiente: guía) con tablas y múltiples comandos para

comprobar que el *proxy* funcione correctamente. Además, nosotros en particular, añadiremos otra forma fácil de chequearlo en el siguiente apartado.

9.4.3. DESPLIEGUE DE LA MALLA DE SERVICIOS

Una vez que hemos inyectado Istio en nuestros archivos o espacio de nombres, vamos a mostrar el despliegue de la aplicación. Este despliegue va a realizarse de la misma manera que en Kubernetes, por tanto, no hay nada nuevo que explicar. Ejecutamos el comando que aplique los archivos de configuración y comprobamos que están corriendo adecuadamente.

```
C:\Users\rauld\Documents\GitHub\TFG\Proyecto-TFG\istio>kubectl get all
NAME                                      READY   STATUS    RESTARTS   AGE
pod/empleado-jdbc-v1-5db9f99cb7-qq8s9     2/2     Running   0          81s
pod/empleado-jdbc-v2-559999c579-gfv9q     2/2     Running   0          68s
pod/my-php-web-app-5687dbbc9-7v7nc        2/2     Running   0          58s
pod/mysqldb-84ff466767-4qx6s              2/2     Running   0          4m29s
pod/python-jdbc-v1-54b75fcfdc-frzqb       2/2     Running   1          39s
pod/python-jdbc-v2-c665cd5bc-q6t6s        2/2     Running   1          29s
pod/usuario-jdbc-5fd787674c-wghfz         2/2     Running   0          16s

NAME                      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/empleado          NodePort    10.110.194.28    <none>        8080:30555/TCP   73s
service/kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP          3d21h
service/my-php-web-app    NodePort    10.111.83.111    <none>        80:31193/TCP     50s
service/mysqldb           NodePort    10.109.38.121    <none>        3306:30197/TCP   3m5s
service/python            NodePort    10.100.149.157   <none>        4000:30094/TCP   33s
service/usuario-jdbc      NodePort    10.109.158.136   <none>        8080:32657/TCP   10s

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/empleado-jdbc-v1     1/1     1            1           81s
deployment.apps/empleado-jdbc-v2     1/1     1            1           68s
deployment.apps/my-php-web-app       1/1     1            1           58s
deployment.apps/mysqldb              1/1     1            1           4m29s
deployment.apps/python-jdbc-v1       1/1     1            1           39s
deployment.apps/python-jdbc-v2       1/1     1            1           29s
deployment.apps/usuario-jdbc         1/1     1            1           16s

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/empleado-jdbc-v1-5db9f99cb7     1         1         1       81s
replicaset.apps/empleado-jdbc-v2-559999c579     1         1         1       68s
replicaset.apps/my-php-web-app-5687dbbc9        1         1         1       58s
replicaset.apps/mysqldb-84ff466767              1         1         1       4m29s
replicaset.apps/python-jdbc-v1-54b75fcfdc       1         1         1       39s
replicaset.apps/python-jdbc-v2-c665cd5bc        1         1         1       29s
replicaset.apps/usuario-jdbc-5fd787674c         1         1         1       16s
```

Figura 95- Despliegue de la aplicación en Istio

Si comparamos esta figura con la del despliegue en Kubernetes, podremos observar algunos cambios, aparte de los evidentes en la etiqueta “AGE”, como ya adelantamos los contenedores de los *pods* indican que son dos por unidad y otro cambio al margen de la configuración de Istio, es que ha ocurrido un reinicio en dos de los *pods* que nos permite comprobar que los *deployments* y los *replicasets* están funcionando bien, y han podido paliar es error en el despliegue.

Pasemos a introducir una manera fácil de comprobar que el proxy está funcionando correctamente. Al tener dos versiones en algunos servicios podemos comprobar el balanceo de carga entre estas versiones. Si accedemos a la interfaz gráfica y pulsamos en la opción del menú “Empleados”, veremos que la última funcionalidad nos permite saber a qué versión estamos accediendo en todas las peticiones. En Kubernetes si accedíamos a este apartado y recargábamos la página podíamos observar que la cantidad de peticiones a cada versión era aproximadamente 50%, pero en orden

alterno, es decir, no seguía un patrón estricto en los accesos. En cambio, en Istio, sin utilizar ninguna regla o archivo de configuración adicional, podemos observar que el tráfico es repartido exactamente al 50% y que sigue el patrón "1010101010…" siendo "1" la versión 1.0 y "0" la versión 2.0.

Figura 96- Pantalla de la interfaz, versiones Empleado

9.4.4. FUNCIONALIDADES DE LA MALLA DE SERVICIOS

A continuación, se explican las funcionalidades implementadas de Istio.

GESTIÓN DEL TRÁFICO – FLUJO

El punto de entrada de la aplicación será a través de internet mediante peticiones que se envían al *Ingress Gateway* el cual se lo pasa al *VirtualService*, este se encarga de enrutar la petición al servicio que corresponda conociendo de esta manera el *DestinationRule*, que nos permite ver la configuración de seguridad, así como la funcionalidad del *Cirtcuit Breaking*.

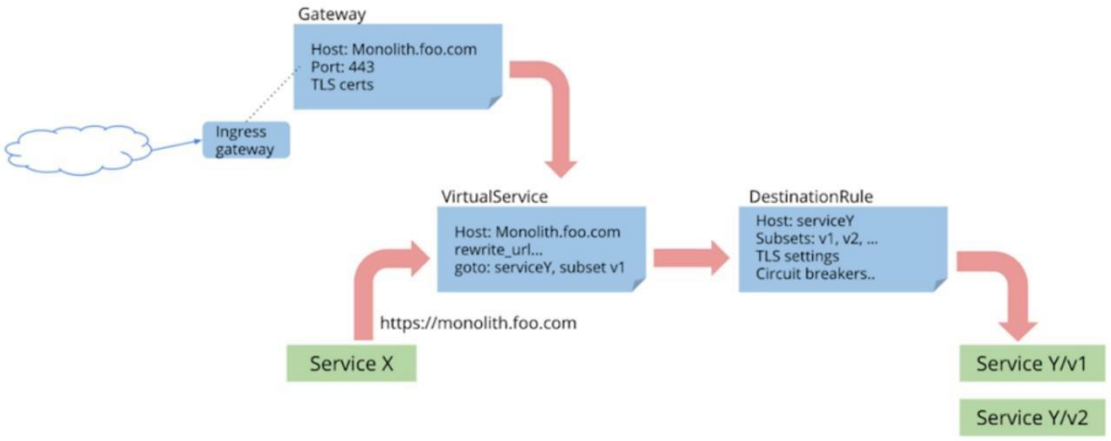


Figura 97- Gestión del tráfico en Istio

GESTIÓN DE POLÍTICAS CON MIXER

Mixer proporciona una capa de intermediación entre los componentes de Istio, así como los servicios basados en Istio, y las copias de seguridad de infraestructura utilizadas para realizar comprobaciones de control de acceso y captura de telemetría. Esta capa permite a los operadores tener una visión amplia y control sobre el comportamiento del servicio sin requerir cambios en los binarios del servicio.

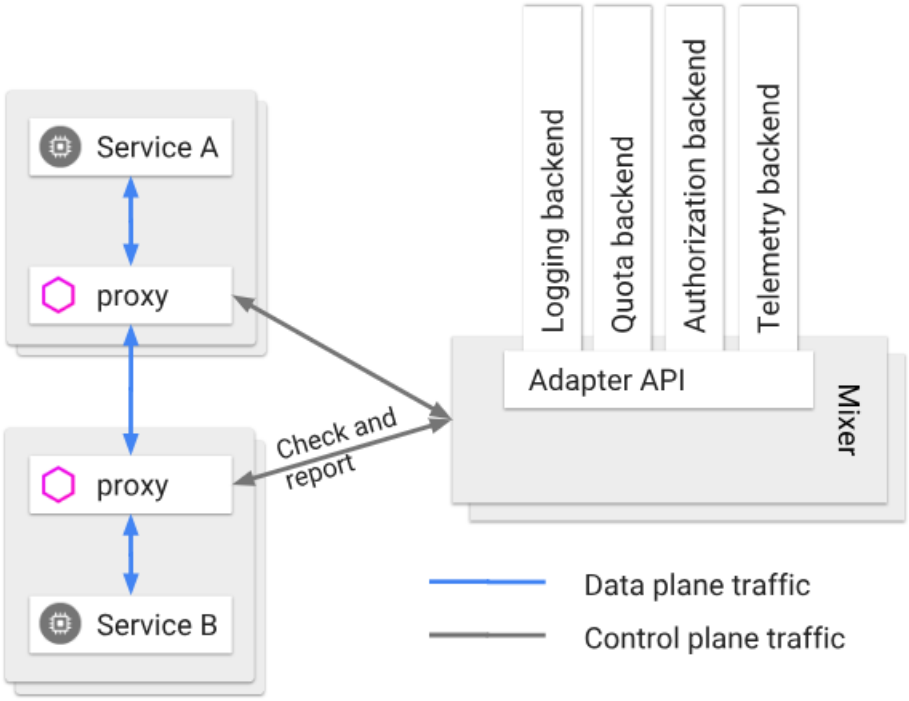


Figura 98- Gestión de las políticas en Istio

GATEWAY Y VIRTUALSERVICES

Un *VirtualService* permite configurar cómo se enrutan las solicitudes a un servicio dentro de una malla, basándose en la conectividad básica y el descubrimiento proporcionados por Istio y su plataforma. Cada servicio virtual consta de un conjunto de reglas de enrutamiento que se evalúan en orden, lo que permite a Istio hacer coincidir cada solicitud dada con el servicio virtual con un destino real específico dentro de la malla.

Se crea una puerta de enlace la cual se mapea a un *VirtualService* que está relacionado con el servicio "my-php-web-app".

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: jdbc-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: jdbc
spec:
  hosts:
  - "*"
  gateways:
  - jdbc-gateway
  http:
  - match:
    - uri:
        prefix: /
    route:
    - destination:
        host: my-php-web-app
```

Figura 99- Implementación del Gateway y de su VirtualService

En la imagen de la izquierda tenemos definido el archivo "gateway.yml" el cual contiene la configuración del punto de entrada de la aplicación, que se indica en el selector `istio: ingressgateway`, así como el puerto 80 donde se expone la aplicación y el protocolo que en este caso es HTTP.

En la imagen de la derecha está definido el *VirtualService*, el cual será el encargado de enrutar las distintas peticiones que le lleguen. En el archivo se define el Gateway a usar el cual es "jdbc-gateway" que se definió previamente. Se indica que todas las peticiones que lleguen a la malla de servicios desde el exterior mediante el prefijo "/" se enrutan al host "my-php-web-app".

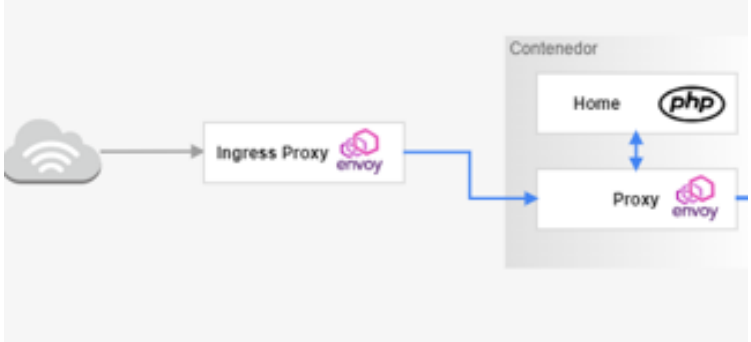


Figura 100- Ingress Proxy de Istio

Todas las peticiones que lleguen al *Ingress Proxy* el cual se le ha configurado una puerta de enlace, se mapea al *VirtualService*, que pasa la petición con prefijo "/", llaman al Envoy del servicio "my-php-web-app" e internamente Kubernetes lo pasa al *pod* correspondiente.

Para saber cuál es la dirección IP en la que esta publicada el *Ingress Proxy* se debe ver los servicios que tiene Istio dentro del clúster de Kubernetes. Para identificarla se busca la línea "istio-ingressgateway".

Con el siguiente comando:

```
$ kubectl get service -n istio-system
```

```
PS C:\Users\rdcabrera> kubectl get service -n istio-system
NAME                     TYPE           CLUSTER-IP       EXTERNAL-IP
grafana                  ClusterIP      10.107.130.228   <none>
istio-egressgateway      ClusterIP      10.100.101.155   <none>
istio-ingressgateway     LoadBalancer   10.110.104.234   localhost
istio-pilot              ClusterIP      10.108.140.163   <none>
istiod                   ClusterIP      10.111.211.107   <none>
jaeger-agent             ClusterIP      None             <none>
jaeger-collector         ClusterIP      10.96.126.147    <none>
jaeger-collector-headless ClusterIP     None             <none>
jaeger-query             ClusterIP      10.106.153.69    <none>
kiali                    ClusterIP      10.111.9.129     <none>
prometheus               ClusterIP      10.105.215.251   <none>
tracing                  ClusterIP      10.97.106.17     <none>
zipkin                   ClusterIP      10.97.121.155    <none>
```

Figura 101- Dirección IP del Ingress Proxy

En la imagen se observa los diferentes servicios que Istio instala en el clúster de Kubernetes. No obstante, lo más importante es el "istio-ingressgateway" y su "EXTERNAL-IP". Se aprecia, que la dirección IP de acceso a la red es localhost. La configuración se realizará a través de los recursos de la puerta de enlace y *VirtualService*.

Con el siguiente comando se puede ver el Gateway creado:

```
$ kubectl get Gateway
```

```
PS C:\Users\rdcabrera> kubectl get gateway
NAME           AGE
jdbc-gateway   21m
```

Figura 102- Gateway por consola

ENRUTAMIENTO INTELIGENTE

El enrutamiento inteligente es una de las principales características que nos ofrece la malla de servicios que nos permite de forma eficaz controlar las peticiones que nos llegan a la aplicación, así como desplegar nuevos microservicios. A continuación, se explican las funcionalidades de enrutamiento aplicadas.

BLUE GREEN DEPLOYMENT

Es una implementación de enrutado que permite cambiar el tráfico, dirigiéndolo hacia solo una versión. Se requiere varias versiones de un mismo servicio. Esta funcionalidad es muy útil, ya que nos permite modificar una versión sin que afecte al funcionamiento de la aplicación.

Para poder enrutar por versión, se debe crear un *VirtualService* para asociarlo con el recurso del microservicio que se quiere mapear, por el que pasara el tráfico de las peticiones. Además, se debe crear una *DestinationRule* donde se indicarán las configuraciones del enrutamiento.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: empleado
spec:
  hosts:
  - empleado
  http:
  - route:
    - destination:
        host: empleado
        subset: version-v1
      weight: 100
```

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: empleado
spec:
  hosts:
  - empleado
  http:
  - route:
    - destination:
        host: empleado
        subset: version-v2
      weight: 100
```

Figura 103- Implementación de Blue Green Deployment en Istio

En la imagen anterior se observa dos dicheros YAML muy parecidos. Estos ficheros son los *VirtualService* definidos para el servicio "Empleado", con esta configuración se consigue que, al aplicarla sobre todo el tráfico, se dirija solo hacia una única versión. Si, por ejemplo, decidimos que todo el tráfico se dirija hacia la versión 2, deberíamos ejecutar el archivo que se encuentra a la derecha.

Para el correcto funcionamiento, se debe crear el *DestinationRule* el cual va a consultar el *VirtualService* previamente creado y saber a qué servicio llamar mediante la etiqueta "subset".

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  creationTimestamp: null
  name: empleado
spec:
  host: empleado
  subsets:
  - labels:
      version: v1
    name: version-v1
  - labels:
      version: v2
    name: version-v2
```

Figura 104- Implementación de DestinationRule empleado Istio

En esta imagen se aprecia el archivo de configuración *DestinationRule*. En este archivo se especifica que el servicio empleado tiene 2 versiones "v1" y "v2", que se definen en la etiqueta "labels". Esta configuración permite que el *VirtualService* obtenga la información necesaria para poder enrutarlo.

Con el siguiente comando se aplica el V*irtualService:*

```
$ kubectl apply -f empleado-v1.yml
```

Para verificar que se ejecutó correctamente se verifica que el V*irtualService* aparece en el listado, como se muestra a continuación.

```
PS C:\Users\rdcabrera> kubectl get virtualservice
NAME       GATEWAYS          HOSTS        AGE
empleado                     [empleado]   13m
jdbc       [jdbc-gateway]    [*]          22m
```

Figura 105- VistualServices activos

Para verificar que el *DestinationRule* se aplicó correctamente, debemos ejecutar el comando que se muestra en la siguiente figura:

```
PS C:\Users\rdcabrera> kubectl get destinationrule
NAME       HOST       AGE
empleado   empleado   111m
python     python     111m
```

Figura 106- DestinationRules activas

Si accedemos a través de la web y realizados varias peticiones al servicio empleado, este siempre será la versión 1, ya que ha sido la que aplicamos en el primer comando de esta página.

Figura 107- Pruebas de Blue Green Deployment

CANARY RELEASES

La implementación de enrutado *Canary* es una implementación que comienza con la actualización de un solo nodo o un subconjunto de ellos. De esta manera, se puede probar la actualización en una pequeña parte del sistema antes de actualizar cada servicio. Normalmente se usa para introducir poco a poco una nueva versión en el sistema.

En este proyecto, lo que se hará es enviar un pequeño porcentaje a una versión del servicio "Empleado", por ejemplo, "v2" y el otro porcentaje a la versión "v1". Empezamos probando la configuración de peso "95-5". Para implementar el *Canary* se requiere de varias versiones del mismo servicio.

Se debe crear un *VirtualService* y su archivo *DestinationRule*, donde se indican las distintas versiones.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: empleado
spec:
  hosts:
  - empleado
  http:
  - route:
    - destination:
        host: empleado
        subset: version-v1
      weight: 95
    - destination:
        host: empleado
        subset: version-v2
      weight: 5
```

Figura 108- Implementación de una Canary Releases en Istio

En la imagen anterior se observa el *VirtualService* definido para el servicio "Empleado", y sigue las pautas expuestas anteriormente, pasando el 95% del tráfico a la versión 1.

El *DestinationRule* es el mismo definido en el apartado del *Blue Green*.

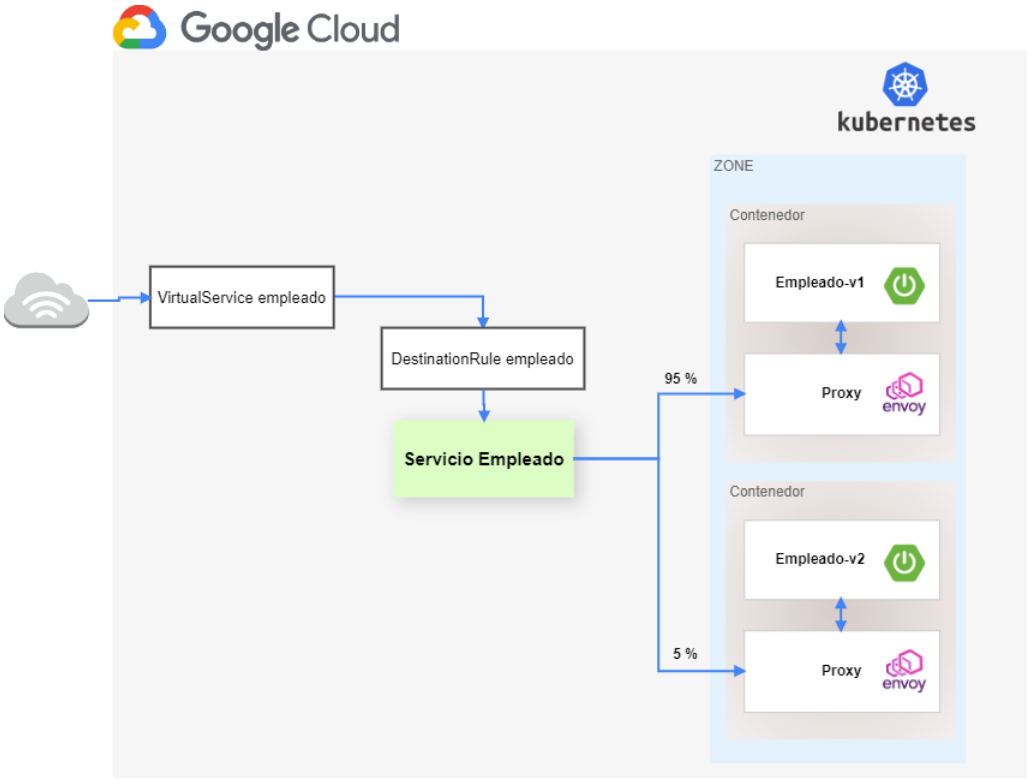


Figura 109- Canary release

El flujo es el siguiente, cualquier petición que llegue al servicio empleado primero pasa por el *VirtualService* definido, el que se encarga de decidir a qué servicio enruta la petición. Para saber el servicio consulta al archivo *DestinationRule*, una vez que tiene el servicio llama al *pod* de Kubernetes.

Esta funcionalidad nos da muchos beneficios, ya que si queremos añadir una nueva versión podemos verificar que funcione correctamente, así como su fiabilidad, y si afecta al funcionamiento de la aplicación.

DARK LAUNCHES

Es una funcionalidad que permite desviar el tráfico a dos versiones a la vez, es decir, el usuario vería el comportamiento de la versión 1, pero los desarrolladores podremos conocer el comportamiento de la versión 2 al recibir la misma petición que la versión 1. Esto nos ayuda a decidir si la versión 2 está preparada para ser lanzada. Se requiere varias versiones de un mismo servicio.

Se debe crear un *VirtualService* y su archivo *DestinationRule*, donde se indican las distintas versiones.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: empleado
  namespace: default
spec:
  hosts:
  - empleado
  http:
  - route:
    - destination:
        host: empleado
        subset: version-v1
    mirror:
      host: empleado
      subset: version-v2
```

Figura 110- Implementación de un Dark Launch

En la imagen anterior se observa el *VirtualService* definido para el servicio “Empleado”, gracias a la etiqueta “mirror” podemos enviar una petición a la versión 1 (que es con la que interactuará el usuario), y replicarla a la versión 2 (podremos observar el comportamiento de la versión 2 a través de los *logs* de la aplicación).

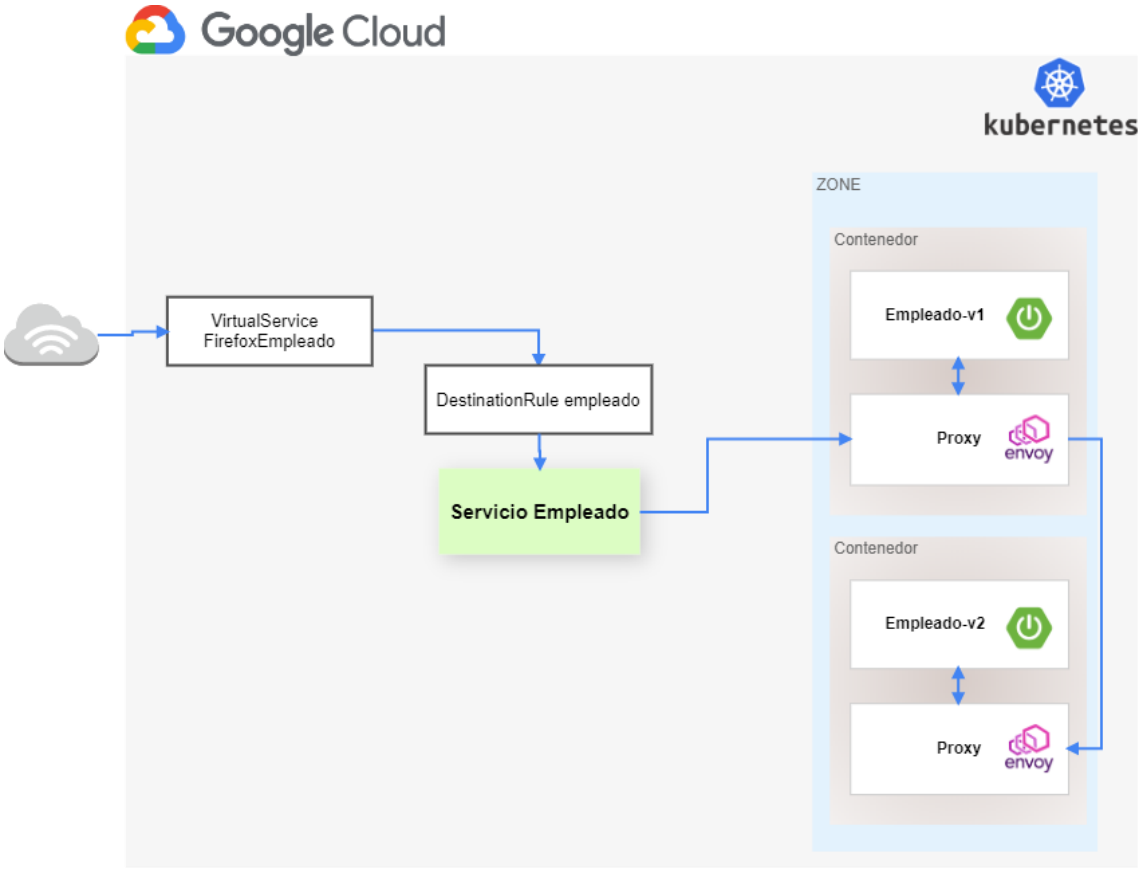


Figura 111- Dark Launch

El flujo es el siguiente, cualquier petición que llegue al servicio empleado primero pasa por el *VirtualService* definido, el que se encarga de decidir a qué servicio enruta la petición. Para saber el servicio consulta al archivo *DestinationRule*, una vez que tiene el servicio llama al *pod* de Kubernetes y este a su vez llama a la versión 2 de empleado. De esta manera, conseguimos probar la versión 2 del servicio sin afectar el funcionamiento del sistema.

Esta funcionalidad nos proporciona una gran ventaja en el escalado de nuestra aplicación, ya que si queremos añadir una nueva versión podemos verificar que funcione correctamente si necesidad de modificar el sistema.

LOAD BALANCER

Es un *VirtualService* que decide el peso del tráfico que recae sobre cada versión, la configuración de estos archivos es igual que la de los *Canary Releases*, pero se usan en otro contexto. Se requiere varias versiones de un mismo servicio.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: empleado
spec:
  hosts:
  - empleado
  http:
  - route:
    - destination:
        host: empleado
        subset: version-v1
      weight: 50
    - destination:
        host: empleado
        subset: version-v2
      weight: 50
```

Figura 112- Implementación de equilibrio de carga

En la imagen anterior se observa el *VirtualService* definido para el servicio “Empleado”, como ya sabemos Istio por defecto configura el tráfico sobre dos versiones siguiendo el patrón “1010101010…”, con esta configuración conseguimos también establecer el tráfico al 50% pero sin seguir ningún patrón.

El siguiente diagrama específica el comportamiento que permite esta funcionalidad.

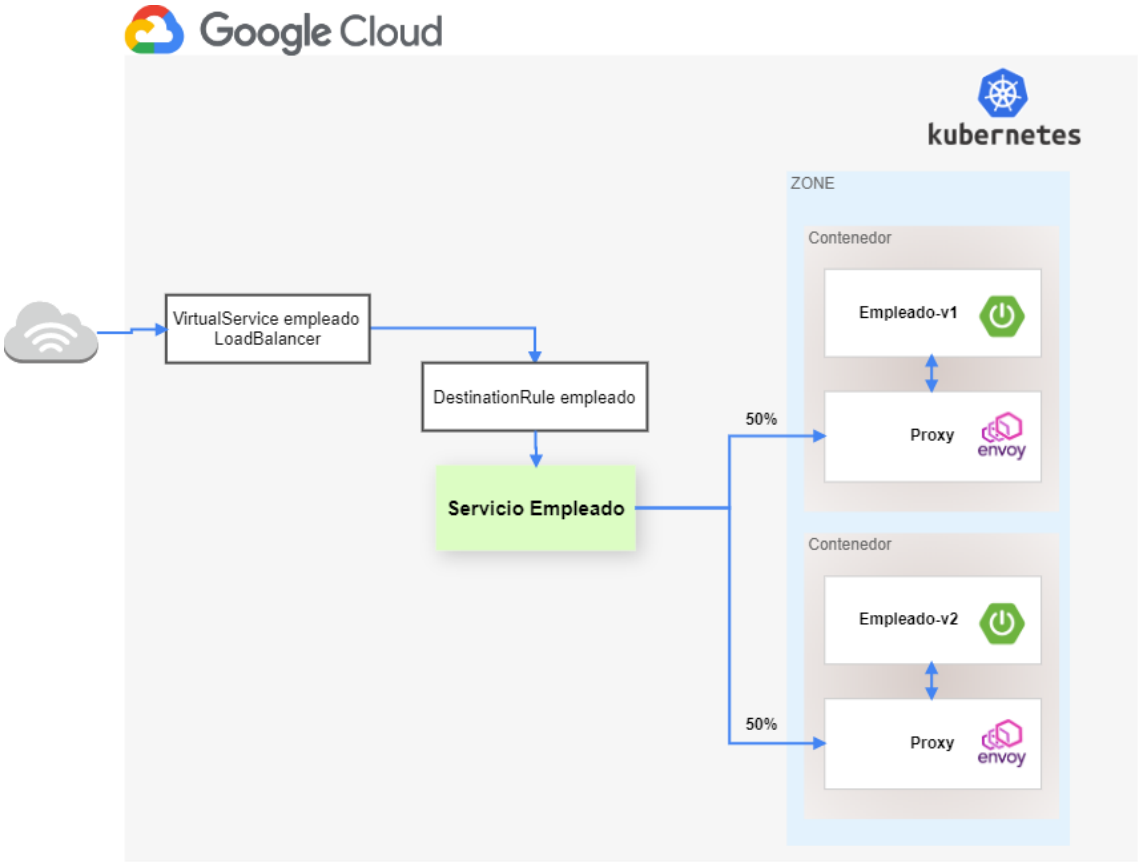


Figura 113- Equilibrio de Carga

TRÁFICO POR CONTENIDO

Se utiliza para enviar el tráfico a distintas versiones dependiendo de su contenido como, por ejemplo, diferenciar entre un usuario que accede desde Chrome, a uno que lo haga desde Firefox. Se requiere varias versiones de un mismo servicio.

De esta manera, se puede enrutar las peticiones a un servicio u otro. Se debe crear su *VirtualService* correspondiente.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  creationTimestamp: null
  name: empleado
spec:
  hosts:
  - empleado
  http:
  - match:
    - headers:
        baggage-user-agent:
          regex: .*Firefox.*
    route:
    - destination:
        host: empleado
        subset: version-v2
  - route:
    - destination:
        host: empleado
        subset: version-v1
```

Figura 114- Implementación de tráfico por contenido

Como vemos en la imagen anterior se observa el *VirtualService* definido para el servicio “Empleado”, con esta configuración se consigue que la petición se enrute de acuerdo con su cabecera si el campo “user-agent” es igual a Firefox o no.

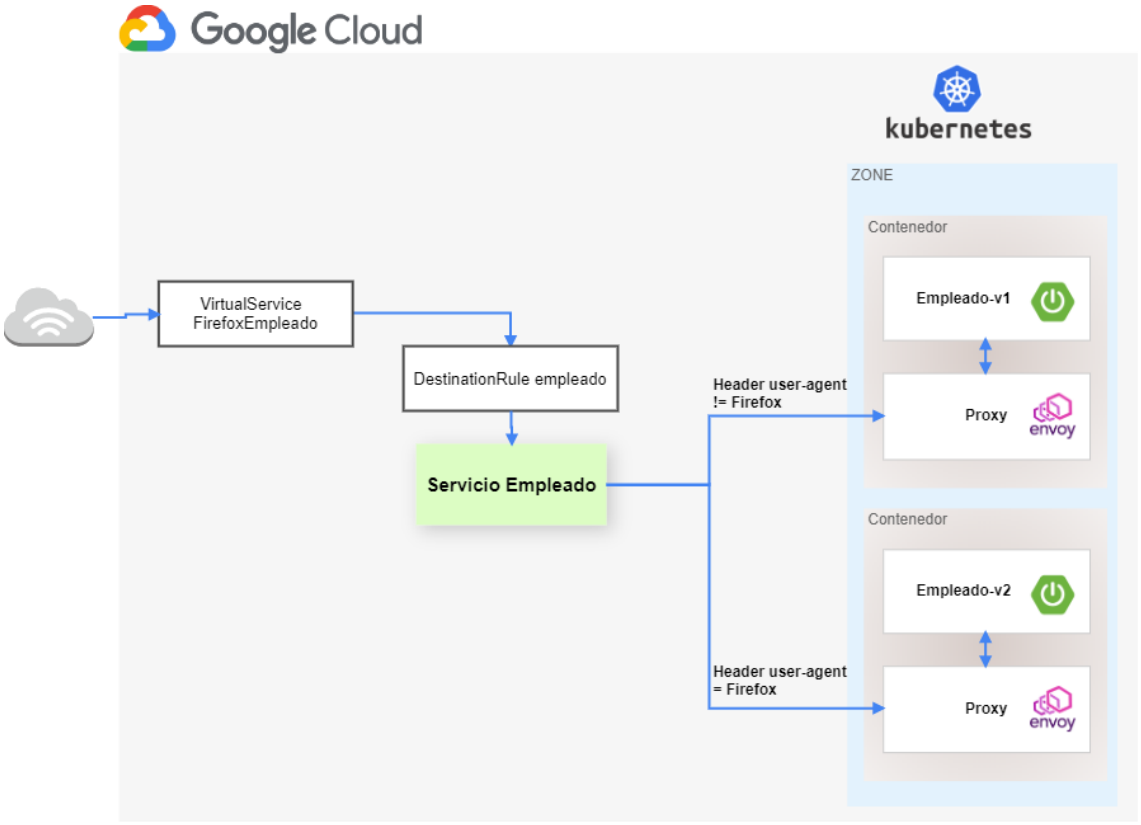


Figura 115- Tráfico por contenido

El flujo es el siguiente, cualquier petición que llegue al servicio empleado primero pasa por el *VirtualService* definido, dependiendo de la cabecera "user-agent" que se envíe, enruta la petición al servicio v1 en caso de que sea distinto de Firefox o al servicio v2 en caso de que sea igual a Firefox.

Esta funcionalidad es muy interesante, ya que podemos enviar las peticiones dependiendo de contenido de la solicitud, o desde que navegador accede; incluso se puede verificar si viene de Android o iOS, lo que es muy útil para probar la funcionalidad de los servicios.

CIRCUIT BREAKER

Detecta cuando una versión esta sobresaturada de peticiones, y dirige el tráfico a otra versión, para que se solvente esta sobresaturación. Se requiere varias versiones de un mismo servicio.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  creationTimestamp: null
  name: empleado
  namespace: default
spec:
  host: empleado
  subsets:
    - name: version-v1
      labels:
        version: v1
    - name: version-v2
      labels:
        version: v2
      trafficPolicy:
        connectionPool:
          http:
            http1MaxPendingRequests: 1
            maxRequestsPerConnection: 1
          tcp:
            maxConnections: 1
        outlierDetection:
          baseEjectionTime: 120.000s
          consecutiveErrors: 1
          interval: 1.000s
          maxEjectionPercent: 100
```

Figura 116- Implementación de Circuit Breaker

Como podemos observar, detecta las versiones de la aplicación y define varias reglas. Estas dictaminan que cuando el plano de control detecte que en una versión hay peticiones en espera o que el procesado de una petición se está demorando más de lo debido, expulsa del circuito el 100% de las veces a esa versión, es decir, elimina a esta versión el atributo de posible *host*, esto puede ocurrir tanto en HTTP como en TCP. Como resultado el usuario que haga una petición a la versión que está fallando obtendrá un error (en HTTP sería un fallo con código 503).

POOL EJECTION

Esta funcionalidad, se podría decir que es una versión mejorada de la anterior, detecta un posible fallo y determina que no se puedan a hacer más peticiones a ese *host* durante un tiempo determinado.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  creationTimestamp: null
  name: empleado
  namespace: default

spec:
  host: empleado
  subsets:
  - labels:
      version: v1
    name: version-v1
    trafficPolicy:
      connectionPool:
        http: {}
        tcp: {}
      loadBalancer:
        simple: RANDOM
      outlierDetection:
        baseEjectionTime: 15.000s
        consecutiveErrors: 1
        interval: 5.000s
        maxEjectionPercent: 100
  - labels:
      version: v2
    name: version-v2
    trafficPolicy:
      connectionPool:
        http: {}
        tcp: {}
      loadBalancer:
        simple: RANDOM
      outlierDetection:
        baseEjectionTime: 15.000s
        consecutiveErrors: 1
        interval: 5.000s
        maxEjectionPercent: 100
```

Figura 117- Implementación de Pool Ejection

Como vemos este archivo define que cuando haya algún tipo de error abre el circuito el 100% de las veces durante 15 segundos. Es decir, la versión no recibirá más peticiones desde la detección del error hasta los 15 segundos siguientes, en el momento en el que se cumple este tiempo el plano de control vuelve a introducir a este *host*. Esto es muy

útil, ya que podemos definir el tiempo que creamos suficiente para que ese *host* se recupere.

Para aclarar aún más la diferencia con respecto al Circuit Breaker, esta funcionalidad posibilita que en cuanto un usuario recibe un error "503", no se permitá el acceso a esta versión, es decir que no obtendremos más errores "503", a diferencia del Circuit Breaker que sigue permitiendo la aparición de este error, aunque no el acceso a la funcionalidad de esa versión. Esto se consigue escalando la aplicación para poseer dos *pods* por cada versión.

OBSERVABILIDAD

Istio genera telemetría detallada para todas las comunicaciones de servicio dentro de una malla de servicios. Esta telemetría proporciona la capacidad de observación del comportamiento del servicio, lo que permite a los operadores solucionar problemas, mantener y optimizar sus aplicaciones, sin imponer cargas adicionales a los desarrolladores de servicios. A través de Istio, los operadores obtienen una comprensión profunda de cómo interactúan los servicios monitoreados, tanto con otros servicios como con los propios componentes de Istio.

Como se ha explicado anteriormente, el administrador de las métricas es mixer. El componente Mixer es altamente modular y extensible. Una de sus funciones principales es abstraer la información detallada de diferentes sistemas de *back-end* de políticas y telemetría, lo que hace que el resto de Istio sea independiente de esos *back-end.*

KIALI

Kiali es una consola de administración para la malla de servicios basada en Istio. Proporciona paneles, observabilidad y permite operar la malla con capacidades de configuración y validación robustas. Muestra la estructura de la malla de servicios al inferir en la topología del tráfico y muestra el estado de su malla. Kiali proporciona métricas detalladas, validación potente, acceso a Grafana e integración sólida para el rastreo distribuido con Jaeger.

Lo más importante es que Kiali proporciona una vista gráfica interactiva de su espacio de nombres en tiempo real que proporciona visibilidad de características como interruptores de circuito, tasas de solicitud, latencia e incluso gráficos de flujos de tráfico. Kiali ofrece información sobre los componentes en diferentes niveles, desde aplicaciones hasta servicios y cargas de trabajo, y puede mostrar las interacciones con información contextual y gráficos en el nodo o borde gráfico seleccionado.

Para instalar Kiali usaremos la demo de Istio usando el comando istioctl:

```
$ istioctl install --set values.kiali.enabled=true
```

Una vez instalado Kiali se verifica que se está ejecutando con el siguiente comando:

```
$ kubectl -n istio-system get svc kiali
```

```
PS C:\Users\rdcabrera> kubectl -n istio-system get svc kiali
NAME    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)     AGE
kiali   ClusterIP   10.111.9.129   <none>        20001/TCP   143m
```

Figura 118- Comprobación del servicio Kiali

Para abrir la interfaz de usuario de Kiali, se debe ejecutar el siguiente comando en el entorno de Kubernetes:

```
$ istioctl dashboard kiali
```

Para iniciar sesión en la interfaz de usuario de Kiali, introducimos el nombre de usuario y la contraseña almacenados en el secreto de Kiali.

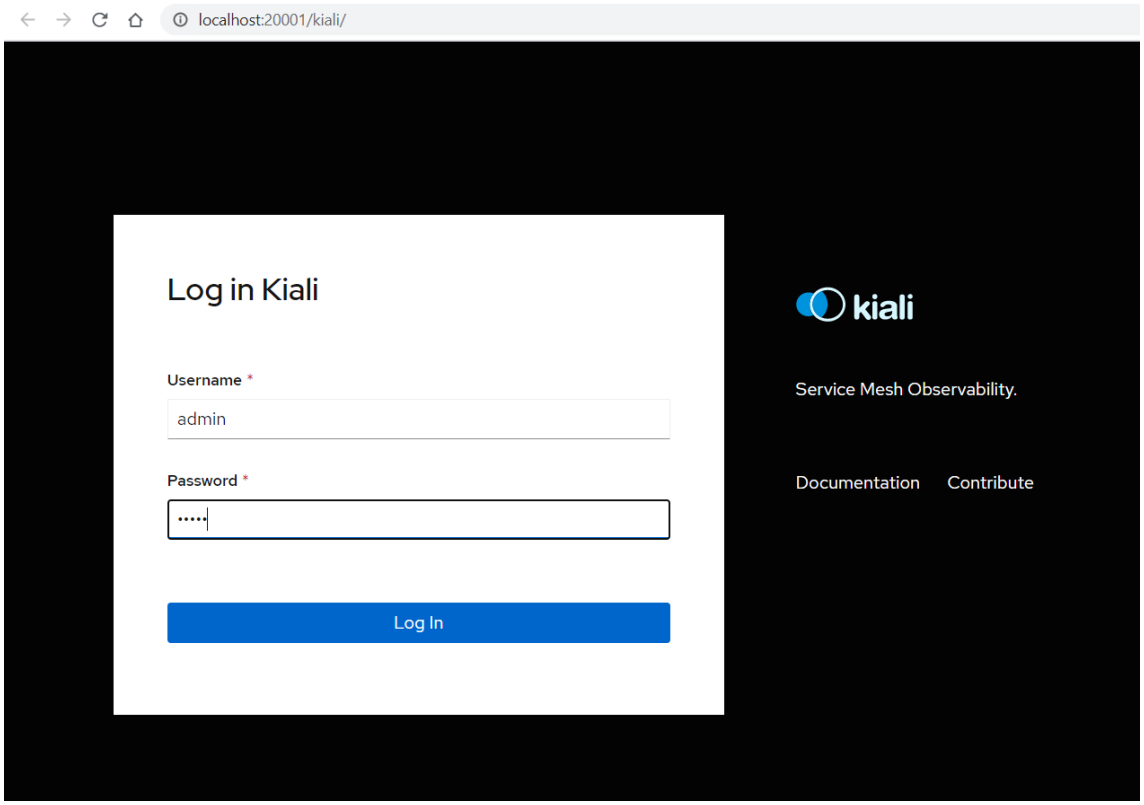


Figura 119- Inicio de sesión en Kiali

A continuación, podemos observar un gráfico de la malla de servicios, incluidos los *VirtualService* y las conexiones entre ellos.

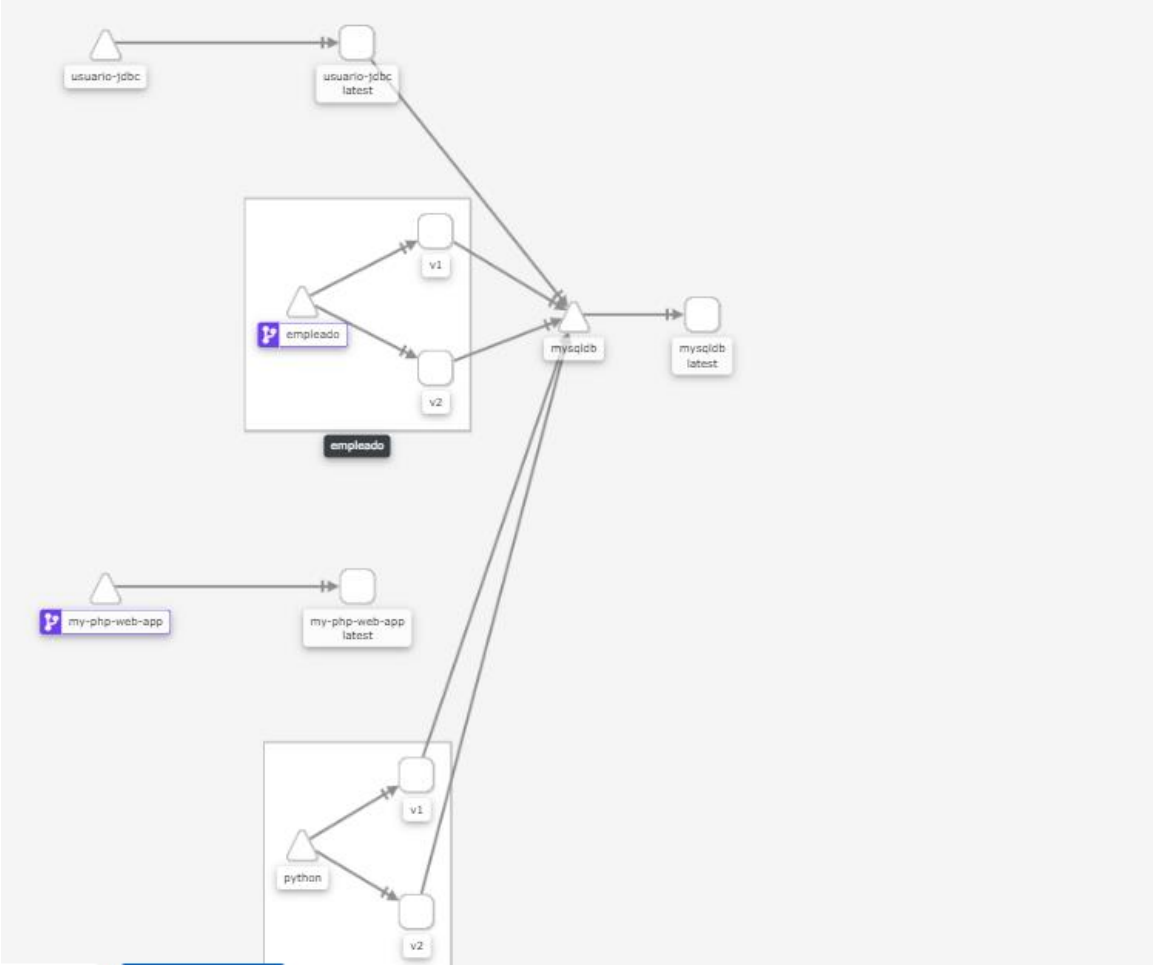


Figura 120- Gráfico de malla de servicios

En la imagen anterior podemos observar todas las conexiones que tenemos en la malla de servicios. Es muy útil ya que podemos ver si las conexiones están fallando y comprobar el porcentaje de tráfico que llega a cada servicio.

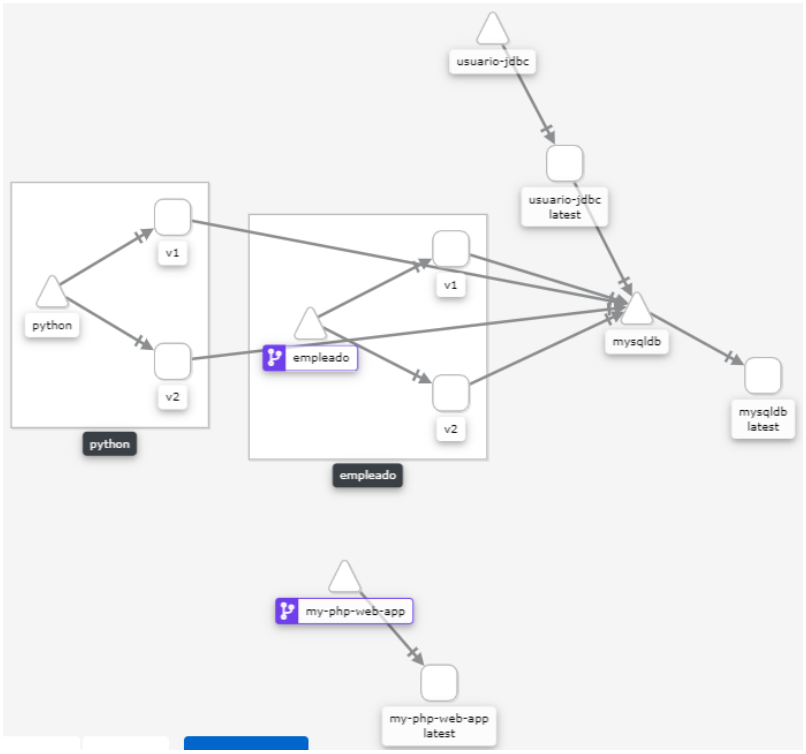


Figura 121- Gráfico de conexiones

Este tipo de gráfico de **Servicio** muestra un nodo para cada servicio de la malla, pero excluye todas las aplicaciones y cargas de trabajo del gráfico.

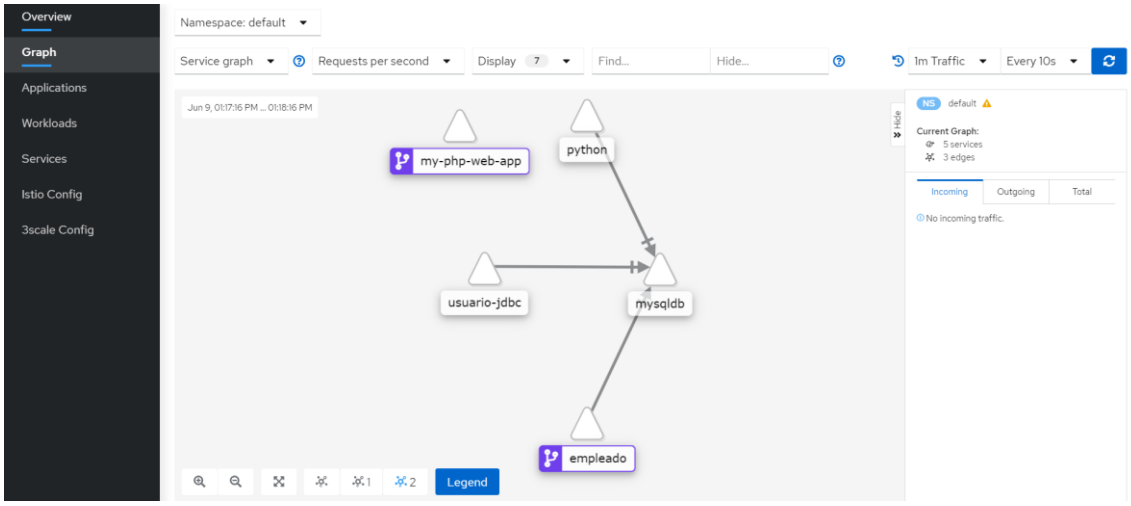


Figura 122- Gráfico de servicios

Este tipo de gráfico de **carga de trabajo** muestra un nodo para cada carga de trabajo en la malla de servicios. Nos es muy útil, ya que podemos ver el flujo de peticiones que pasa por la malla.

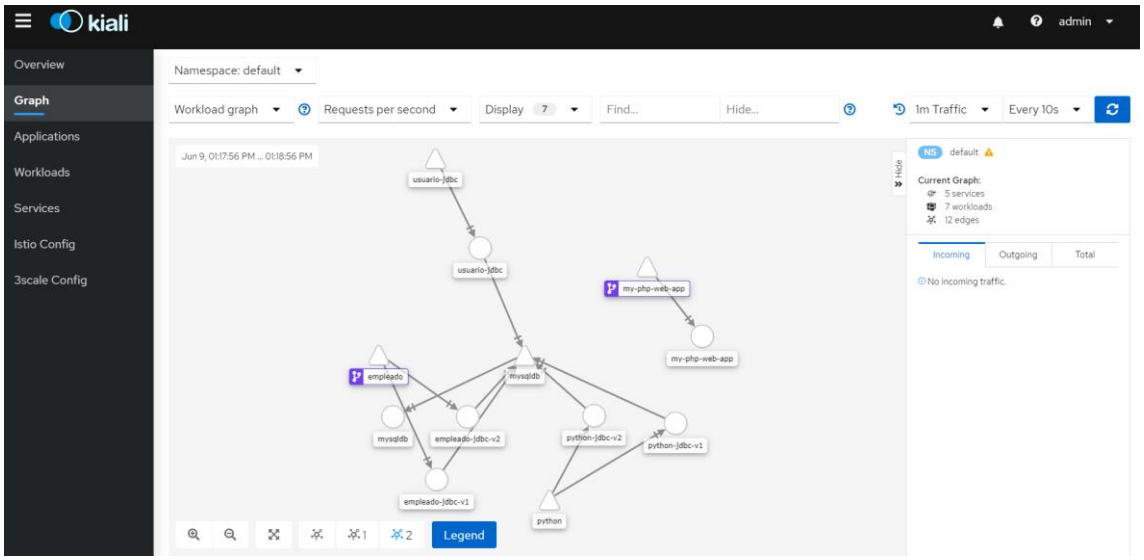


Figura 123- Gráfico de carga de trabajo

PROMETHEUS

Prometheus es una aplicación de software gratuita utilizada para la supervisión y alerta de eventos. Registra métricas en tiempo real en una base de datos de series de tiempo construida utilizando un modelo de extracción HTTP, con consultas flexibles y alertas en tiempo real.

En este caso lo usamos para poder consultar las métricas que nos proporciona Istio. Prometheus ya viene instalado por defecto al usar el perfil demo de Istio. Se verifica que se está ejecutando con el siguiente comando que vemos en la figura:

```
PS C:\Users\rdcabrera> kubectl -n istio-system get svc prometheus
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
prometheus   ClusterIP   10.105.215.251   <none>        9090/TCP   148m
```

Figura 124- Service Prometheus en consola

Con el siguiente comando se abre la interfaz:

```
$ istioctl dashboard prometheus
```

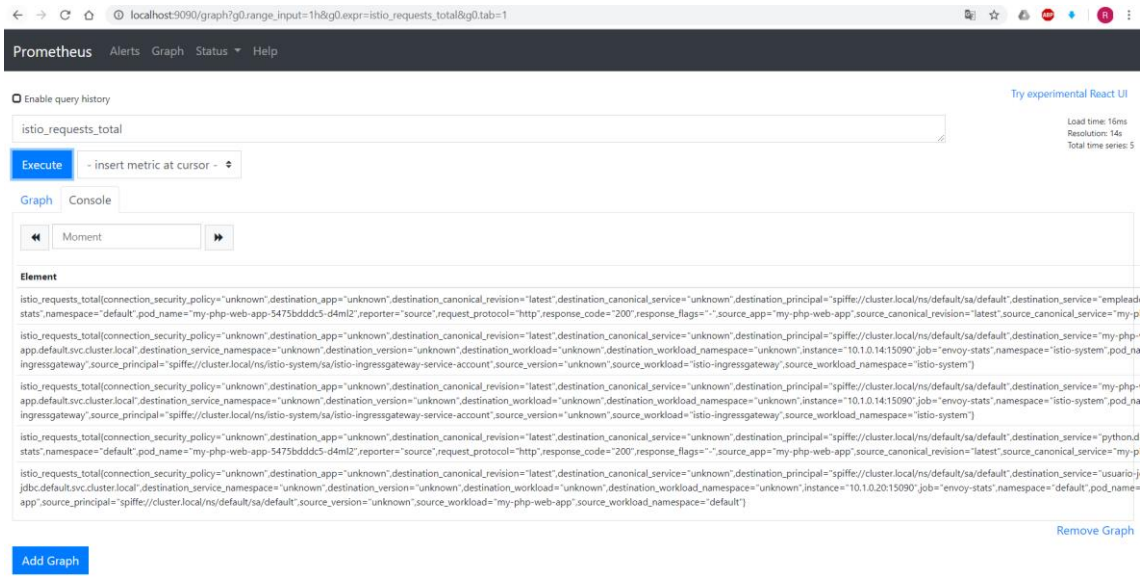


Figura 125- Ejecución de las peticiones totales en Prometheus

Como vemos en la imagen anterior se visualizan las métricas que se poseen al ejecutar la expresión "istio_request_total". Esto no es muy útil, ya que nos permite ver la cantidad de peticiones y tráfico que llega.

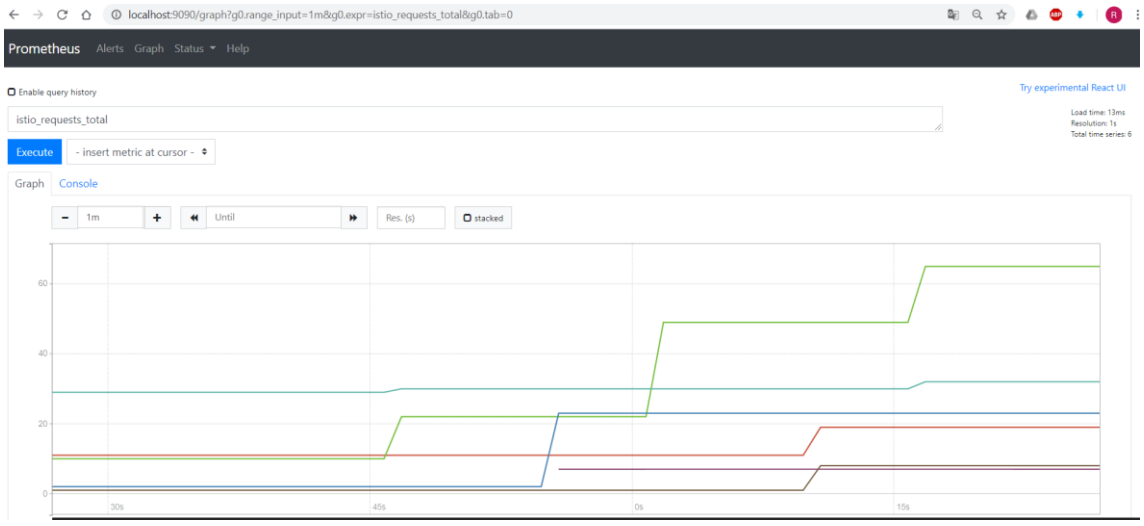


Figura 126- Gráfico peticiones en Prometheus

También podemos ver los resultados de la consulta gráficamente seleccionando la pestaña Gráfico debajo del botón **Ejecutar**.

GRAFANA

Grafana es un software de visualización y análisis de código abierto. Nos permite consultar, visualizar, alertar y explorar las métricas sin importar dónde estén almacenadas.

El uso que le daremos es de observar todo el tráfico que está pasando por nuestra malla de servicios.

Para que Grafana funcione debe estar ejecutándose Prometheus.

Se verifica que el servicio Grafana se esté ejecutando en el clúster con el siguiente comando:

```
PS C:\Users\rdcabrera> kubectl -n istio-system get svc grafana
NAME      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
grafana   ClusterIP   10.107.130.228   <none>        3000/TCP   153m
```

Figura 127- Service Grafana en consola

Para Abrir el panel de control de Istio a través de la interfaz de usuario de Grafana se debe ejecutar el siguiente comando:

```
PS C:\Users\rdcabrera> kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l app=grafana -o jsonpath='{.items[0].metadata.name}') 3000:3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

Figura 128- Panel de control de Istio a través de consola

A continuación, se va a crear tráfico en la malla para visualizarlo en los distintos tableros.

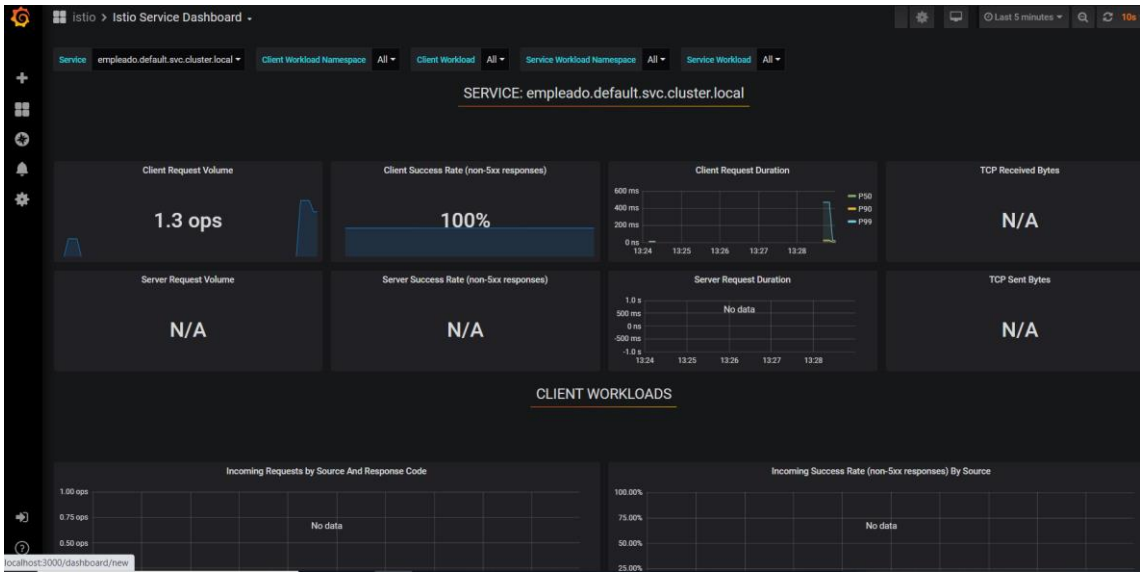


Figura 129- Estadísticas proporcionadas por Grafana

Esto proporciona detalles sobre las métricas para el servicio y luego las cargas de trabajo del cliente (cargas de trabajo que llaman a este servicio) y las cargas de trabajo del servicio (cargas de trabajo que proporcionan este servicio) para ese servicio. Es muy útil, ya que podemos ver donde se están produciendo los errores.

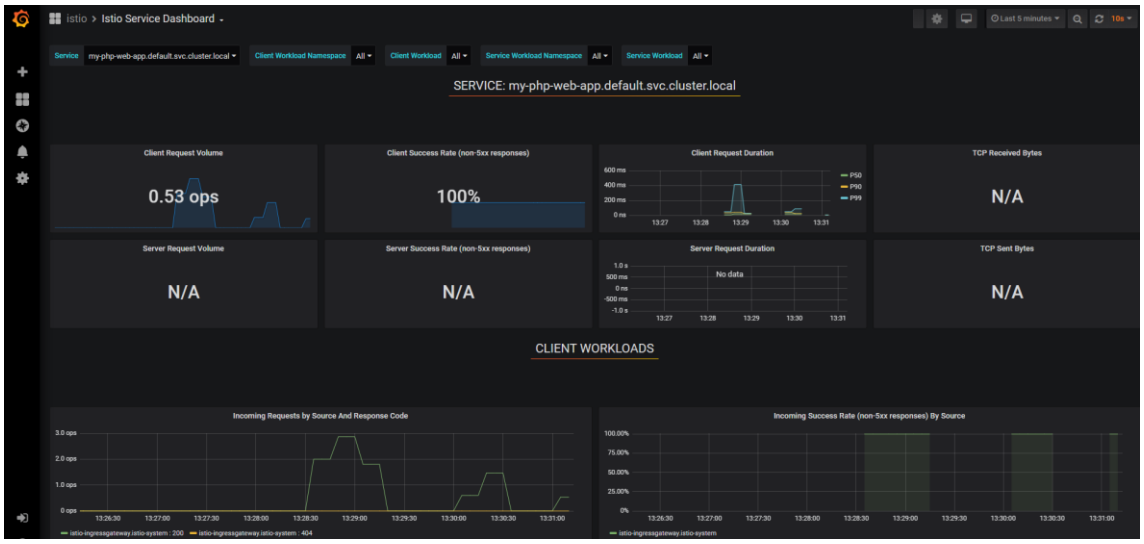


Figura 130- Otras estadísticas en Grafana

Esta figura nos proporciona detalles sobre las métricas para cada carga de trabajo y luego las cargas de trabajo entrantes (cargas de trabajo que envían solicitudes a esta) y los servicios salientes (servicios a los que esta carga de trabajo envía solicitudes) para esta.

Figura 131- Más estadísticas en Grafana

SEGURIDAD

Tener una aplicación orientada a una arquitectura de microservicios nos genera muchos beneficios, agilidad. No obstante, se debe tener en cuenta la seguridad de los servicios ya que tienen requerimientos específicos como son los siguientes:

- Para evitar los ataques de hombre en el medio, es necesario que el tráfico de la red este cifrado.
- Para poder dar un control de acceso a los servicios y que este sea flexible es necesario tener políticas de TLS mutuo.
- Para revisar quién ha realizado cada acción, el momento y la hora, es necesario de una herramienta de revisión. Istio nos proporciona muchas de estas funcionalidades que necesitamos.

TLS MUTUO

Istio utiliza TLS mutuo para pasar de forma segura cierta información del cliente al servidor.

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "default"
spec:
  peers:
  - mtls: {}
```

Figura 132- Implementación TLS Policy

Como se puede observar se ha definido una política de seguridad en donde el nombre es default porque aplica para todos los servicios de la malla.

En este momento, solo el receptor se ha configurado para poder usar el TLS mutuo. Debemos configurar la parte del cliente, para ello crearemos un *DestinationRule* para que pueda usar el TLS mutuo.

```
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "default"
spec:
  host: "*.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

Figura 133- Implementación TLS DestinationRule

En este archivo se está definiendo de tal manera que el host solo este limitado en local, es decir, que solo los servicios van a poder llamarse entre sí en el propio clúster y no de manera externa a ellos. Se especifica el modo del TLS que es ISTIO_MUTUAL.

Si aplicamos esta configuración, lo que conseguimos es que los servicios solo puedan acceder a ellos siempre y cuando tengan activado el TLS. Podemos comprobar que el TLS se ha aplicado correctamente comprobando desde de la consola interna de algún contenedor que el tráfico está cifrado, sino fuera así podemos observar información como el tipo de petición, por ejemplo, GET, el origen y el destino, etc.

9.5. GOOGLE CLOUD

En esta sección vamos a mostrar el procedimiento que seguimos para subir la malla de servicios final a Google Cloud. Esto nos permite acceder a la aplicación desde cualquier dispositivo y no limitar el desarrollo a interacciones locales.

Lo primero que debemos hacer es acceder a la página web Google Cloud Kubernetes y registrarnos en esta, como ya adelantamos disponemos de 300$ por cada participante, este presupuesto se va reduciendo incrementalmente mientras que tengamos activo algún clúster. Una vez registrados podemos crear un clúster en la plataforma, siguiendo las indicaciones de la siguiente pestaña.

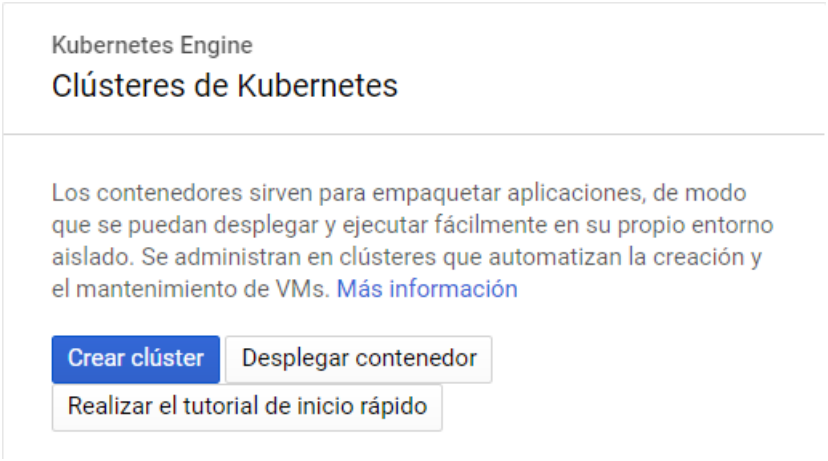


Figura 134- Crear Clúster en Google Cloud

Una vez que pulsamos en el botón “Crear Clúster”, podemos elegir el nombre, la zona y la versión que queremos establecer en nuestro clúster, en la siguiente figura, se muestra la pantalla indicada.

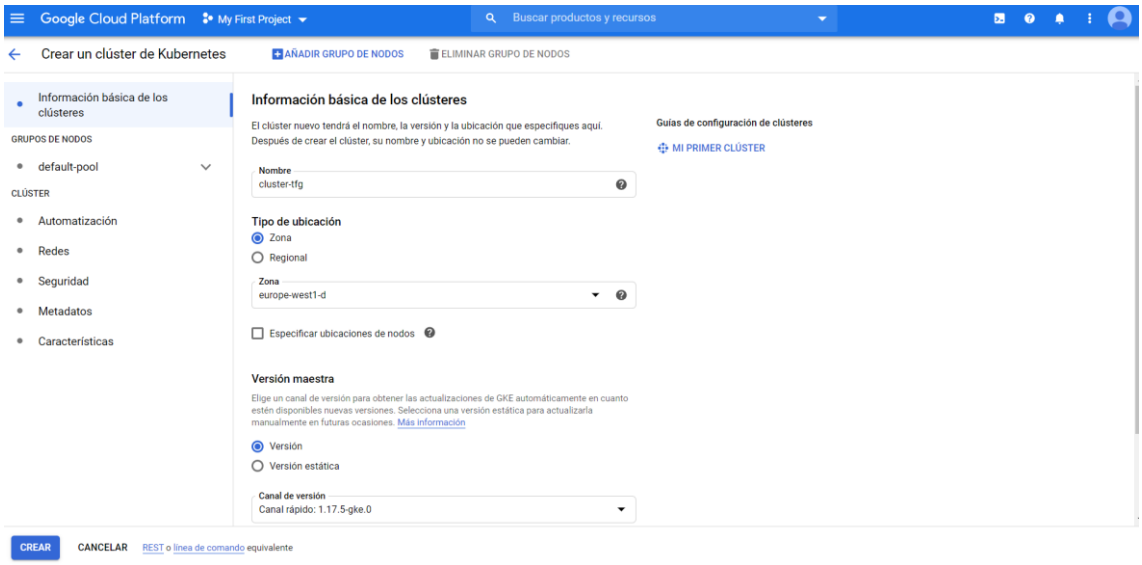


Figura 135- Especificaciones del clúster en Google Cloud

También Google Cloud nos ofrece la opción de conectarnos a nuestro clúster a través de la consola, la siguiente figura muestra el comando que se nos proporciona.

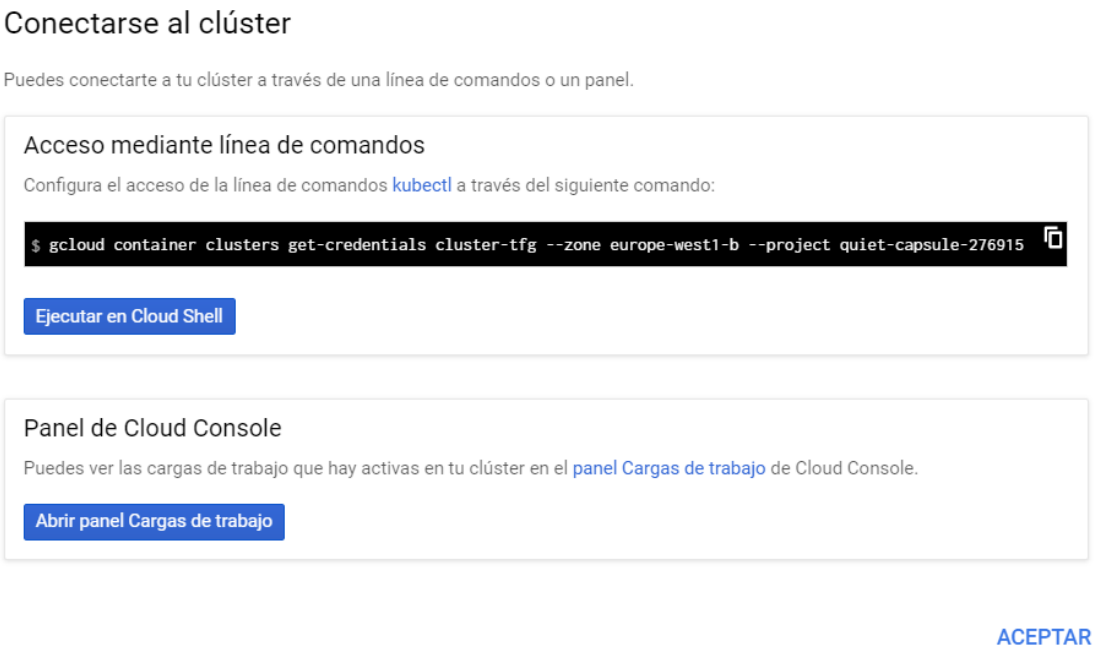


Figura 136- Comando de acceso al clúster en Google Cloud

Una vez que ejecutamos el comando en la Cloud Shell, podemos subir nuestros archivos de configuración con los que ejecutamos nuestra malla de servicios en local. Antes de desplegar la aplicación debemos comprobar que Istio ha sido instalado de forma correcta. Para ello ejecutamos el siguiente comando.

```
rcabre01@cloudshell:~ (quiet-capsule-276915)$ kubectl get service -n istio-system
NAME                       TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                      AGE
istio-citadel              ClusterIP      10.60.11.190   <none>        8060/TCP,15014/TCP
istio-galley               ClusterIP      10.60.13.59    <none>        443/TCP,15014/TCP,9901/TCP
istio-ingressgateway       LoadBalancer   10.60.15.212   <pending>     15020:31964/TCP,80:30041/TCP,4
443:31014/TCP   1s
istio-pilot                ClusterIP      10.60.15.89    <none>        15010/TCP,15011/TCP,8080/TCP,1
istio-policy               ClusterIP      10.60.10.119   <none>        9091/TCP,15004/TCP,15014/TCP
istio-sidecar-injector     ClusterIP      10.60.3.224    <none>        443/TCP                      1s
istio-telemetry            ClusterIP      10.60.2.103    <none>        9091/TCP,15004/TCP,15014/TCP,4
promsd                     ClusterIP      10.60.6.61     <none>        9090/TCP                     1s
```

Figura 137- Services en el namespace "istio-system"

Al subir estos archivos Google Cloud nos permite ejecutarlos en el clúster usando los mismos comandos que hemos expuesto anteriormente. Por tanto, podremos desplegar nuestra aplicación ejecutando el comando:

```
$ kubectl apply -f .
```

Para comprobar que el despliegue ha sido satisfactorio, Google Cloud nos proporciona una URL que sirve como puerta de enlace a nuestra aplicación. En este caso, obtuvimos la dirección 35.205.125.51, la siguiente figura muestra una captura que demuestra que el despliegue y acceso a la dirección ha sido satisfactorio.

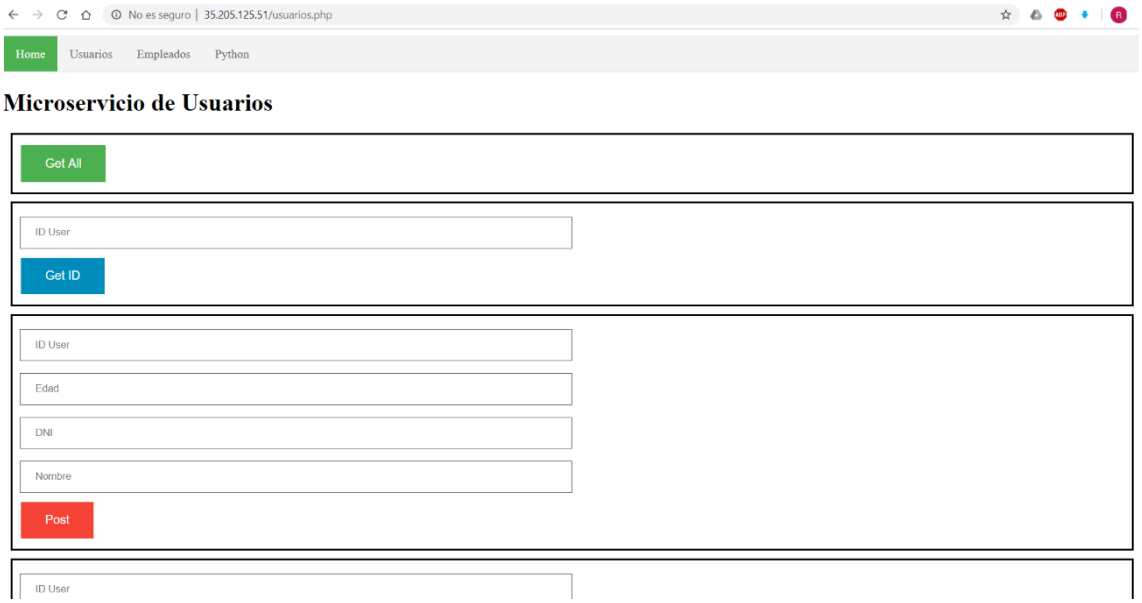


Figura 138- Interfaz de la aplicación accedida utilizando Google Cloud

Con el objetivo, de mostrar más funcionalidades que nos proporciona Google Cloud, vamos a mostrar capturas de las pantallas, que nos han parecido más útiles. La primera muestra la pantalla que usamos para obtener la información general de nuestro clúster.

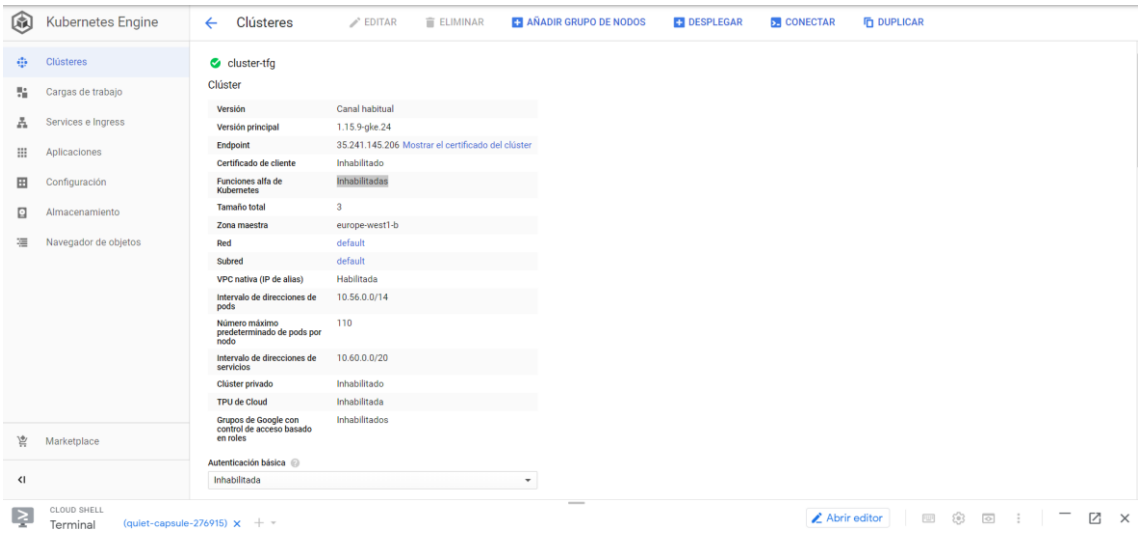


Figura 139- Datos del clúster en Google Cloud

Google Cloud también nos ofrece una pestaña de "Sevices e Ingress" que nos muestran el estado e información (tipo, puerto, punto de conexión, espacio de nombre) de todos los *services* desplegados en nuestro clúster.

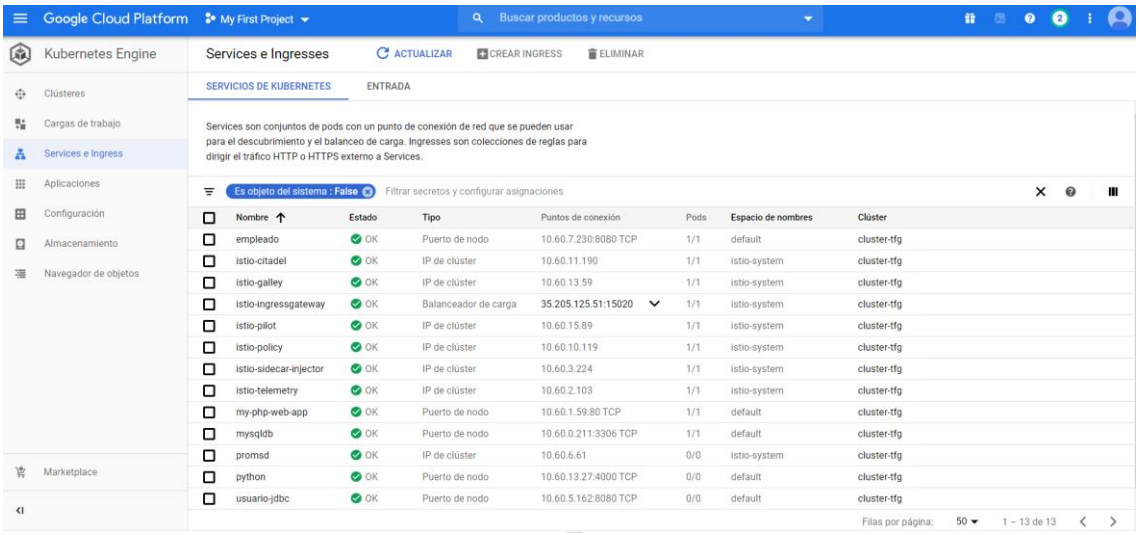


Figura 140- Services e Ingress en Google Cloud

Por último, creemos oportuno mostrar la funcionalidad que nos parece más útil y novedosa con respecto al desarrollo en local. La siguiente figura muestra un resumen de del consumo y peticiones que se han realizado al *service* de la interfaz.

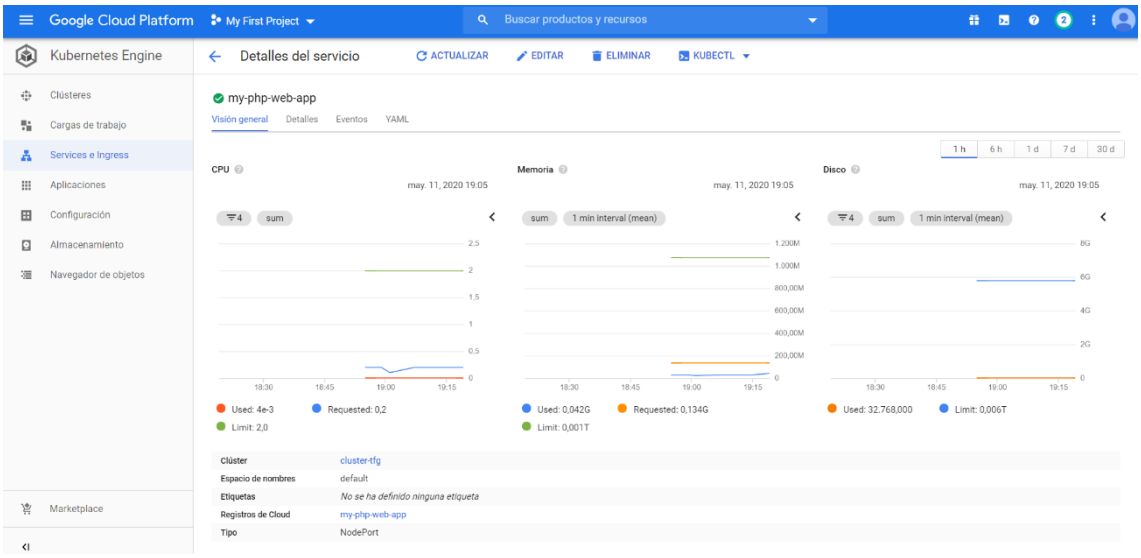


Figura 141- Estadísticas de los accesos a la aplicación en Google Cloud

Los gráficos no nos muestran muchas estadísticas, ya que solo hemos realizado un número limitado de peticiones, aun así, creemos muy útil esta información para cerciorarnos de que la aplicación ha funcionado correctamente en diferentes situaciones.

10. CÓDIGO ASOCIADO

Hemos creado un repositorio en GitHub para la entrega del código realizado durante este proyecto, contiene cada uno de los archivos y funcionalidades explicadas[2].

El repositorio está formado por nueve directorios:

- Empleado: Contiene el código del servicio "Empleado" y su Dockerfile.
- Empleado-v2: Contiene el código de la versión 2.0 del servicio "Empleado" y su Dockerfile.
- Front: Contiene el código de la interfaz y su Dockerfile.
- Istio: Contiene los archivos de configuración de la malla de servicio, los *deployments* y los *services* necesarios para desplegar la aplicación. Además, posee un directorio "Rules" donde se encuentran las *DestinationRules* y los *VirtualServices* explicados en el apartado de implementación de Istio.
- Kubernetes: Posee los archivos de configuración que necesita Kubernetes para desplegar la aplicación, entre ellos se encuentran los *deployments* y los *services* de cada servicio, además, en otros dos directorios se encuentran los *volumes* y el *secret* de la base de datos.
- Python-mysql-v1: Contiene el código del servicio "Python" y su Dockerfile.
- Python-mysql-v2: Contiene el código de la versión 2.0 del servicio "Python" y su Dockerfile.
- SQL: Contiene el archivo de la base de datos, en este caso hemos unido el código del archivo "usuario.sql" y "empleado.sql" para que sea más rápido el despliegue.
- Usuario: Contiene el código de la implementación del servicio "Usuario" y su Dockerfile.

En el caso de que algún usuario desee levantar la aplicación debe descargarse el código. Dando por hecho que el usuario que quiera levantar la aplicación tiene instalado correctamente Docker, Kubernetes e Istio en su ordenador, debe seguir los siguientes pasos:

- Abrir un terminal de la consola en la ruta donde se encuentre el repositorio en local.
- Una vez que se encuentra en la ruta correcta, debe repetir los siguientes comandos en cada directorio que posea un servicio:

```
$ docker build -t <nombre de la imagen> .
```

```
$ docker tag <nombre de la imagen>
<usuario>/<repositorio>:<nombre de la imagen>
```

```
$ docker push <usuario>/<repositorio>:<nombre de la imagen>
```

[2] Se puede acceder a la totalidad del código a través del siguiente enlace: https://github.com/RaulDieg/Entrega-TFG.

- Ya creadas y almacenadas las imágenes en un repositorio personal, que debe haber sido creado anteriormente. Sitúa el terminal en el directorio “kubernetes” y debe cambiar en cada *deployment* el usuario, repositorio e imagen que haya establecido.
- Primero levanta los *volumes* y el *secret* de la base de datos.
- Segundo levanta el *deployment* de la base de datos, y siguiendo las instrucciones del comando “exec” explicadas anteriormente, ejecuta la consola interactiva e introduce el archivo SQL en el servicio. Ejecuta un “exit” para salir.
- Por último, despliega el resto de los archivos de configuración.
- Para desplegar la malla de servicios, debemos seguir los mismos pasos que en el directorio de Kubernetes.
- Para aplicar alguna regla de Istio, seguir el apartado de implementación de este.

11. CONCLUSIONES

En este proyecto hemos implementado una aplicación de microservicios con una malla de servicios, centrándonos en desarrollar y conocer la mayor parte de las funcionalidades que nos ofrece Istio. Es verdad que la implementación de los servicios es básica, pero es suficiente para cubrir los objetivos pautados y aprender las configuraciones de Istio a fondo. Sabiendo esto, vamos a exponer las conclusiones a las que hemos llegado a lo largo de este trabajo.

11.1. VENTAJAS Y DESVENTAJAS

Este apartado hemos decidido hacerlo comparando dos aspectos generales que gracias a la implementación de este proyecto conocemos. La primera comparación y más obvia es la de arquitectura monolítica y de microservicios.

Aspectos	Monolítica	Microservicios
Modularidad	Es una arquitectura rígida y difícilmente adaptable, es decir, al tener toda la aplicación y funcionalidad en la misma capa no se puede desarrollar de forma independiente.	Se basa en servicios autónomos, con gran diversidad de tamaños y funcionalidades, que pueden desarrollarse de forma independiente.
Escalabilidad	Como tiene baja modularidad, tiene que escalarse verticalmente, y puede resultar más costoso.	En contraposición a la monolítica, al tratarse de servicios independientes puede realizarse más rápido y eficazmente.
Versatilidad	Una misma tecnología.	Podemos introducir múltiples lenguajes de programación y trabajar con distintos entornos a la vez.
Diseño	Rápido y unificado, al tratarse de una misma capa el diseño se lleva a cabo en menos tiempo.	Puede llevarse a cabo de varias formas, una de las más usadas es partiendo de una arquitectura monolítica y disgregándola en los distintos servicios. Otra es partiendo de cero, lo cual se hace una tarea ardua y que normalmente necesita de más tiempo.
Implementación	Al ser un sistema de una sola capa las directrices son claras, se pueden ejecutar con menos comunicaciones entre los participantes, pero debe de ser desarrollado con un orden	Como ya decíamos puede llevarse a cabo de forma independiente, aunque las comunicaciones entre los participantes o grupos debe ser continua, con el objetivo

	especifico y no se puede levantar la aplicación hasta que la mayor parte esté desarrollada.	de dirigir la aplicación al mismo destino. Además, se requiere de mayor número de grupos y más especializados, esto puede ser considerado una ventaja o no, dependiendo de cómo se mire, ya que, grupos más especializados permitirán elaborar una aplicación usando menos recursos y abarcando más funcionalidades.
Coste	El coste de implantación es menor.	El coste de implantación es mayor, debido a que, supone un alto coste de infraestructuras y pruebas distribuidas.
Mantenimiento	Suele ser bastante costoso, lo que hace que el ciclo de vida de la aplicación se vea reducido.	Al ser un sistema con alta modularidad el mantenimiento se hace de forma eficiente, además de que los fallos se pueden controlar más fácilmente que en la monolítica.
Producción	Sencilla, y ampliamente conocida.	Sencilla, aunque pueden detectarse errores por falta de comunicación entre desarrolladores.

Tabla 7- Comparación de arquitectura monolítica y de microservicios

En resumen, existen varios aspectos en los que la arquitectura monolítica sigue siendo superior. Lo que nos llevaría a decidir por cual decantarnos serían las características del proyecto y el equipo de desarrolladores.

El siguiente aspecto que debemos analizar es el uso de una malla de servicios frente a una aplicación desarrollada y controlada con un orquestador de contenedores. Como hemos visto a lo largo del trabajo las mallas de servicios, en concreto Istio, abarca bastantes más funcionalidades (*canary releases*, *blue deployment*, autenticación, etc) que un orquestador, además de permitirnos la monitorización con diversas métricas. Como principal desventaja de las mallas de servicios es que aumentan la complejidad de nuestra arquitectura, ya que introducimos elementos como *proxies*, panel de control centralizado o reglas *iptables*. Otra de las desventajas de Istio es que aumenta la latencia en las comunicaciones, aunque puede ser considerado como asumibles, contrastándolo con las múltiples funcionalidades que nos ofrece.

11.2. OPINIONES DE LOS PARTICIPANTES DEL GRUPO

En este apartado, vamos a intentar transmitir las sensaciones que hemos podido sacar tras la elaboración de este proyecto, en concreto, las que prevalecen al finalizar el trabajo.

La principal sensación que nos queda es que una vez que hemos comprendido la arquitectura de microservicios y en particular, la malla de servicios en su totalidad, creemos que es una herramienta muy potente y con un gran futuro en el desarrollo de aplicaciones. Al poder implementar, gran parte de las funcionalidades de Istio, hemos comprendido las múltiples aplicaciones que puede llegar a tener.

En la parte inicial del proyecto, cuando todavía estábamos estudiando el tema, nos dimos cuenta de la cantidad de empresas que han dado el salto a una arquitectura de microservicios (por ejemplo, Netflix), o que están dando los pasos previos para este cambio, y ahora que hemos adquirido los conocimientos suficientes para aportar una opinión objetiva, no nos extraña que se hayan decidido por esta arquitectura. Si es verdad, que la mayoría son grandes empresas con muchos recursos e infinidad de expertos en cada campo. No obstante, aunque no dispongamos de esta cantidad de recursos, elegir esta arquitectura no va a ser un error.

11.3. OBJETIVOS CUMPLIDOS

Al principio de esta memoria, introducimos los objetivos principales y específicos que afrontamos al inicio de este TFG, el objetivo principal era comprender e implementar una arquitectura de microservicios con una malla de servicios.

Creemos que es suficiente información la que hemos conseguido exponer en esta memoria como para poder afirmar que el objetivo principal se ha cumplido. Afrontábamos este trabajo como un reto para comprender una arquitectura con la que no habíamos tenido contacto en ninguna de las asignaturas, y lo terminamos como una opción de futuro, el mundo de la informática es muy amplio, y pocas especialidades o temas poseen la belleza de abarcar conocimientos de dos áreas tan específicas como el desarrollo de aplicaciones y la arquitectura de computadores, es por eso, que quizás en un futuro nos dediquemos a investigar y profundizar más sobre los conocimientos adquiridos.

De los objetivos específicos, cabe destacar que hemos cumplido las expectativas de implementar una aplicación con múltiples lenguajes y plataformas. Gracias a las asignaturas que hemos cursado en la carrera, solo nos ha hecho falta recordar algunas pautas para utilizar lenguajes como Java, Python, SQL o PHP, pero hemos tenido que empezar de cero y profundizar en Spring Boot y la versión Flask de Python.

En resumen, estamos muy contentos con la decisión al escoger este TFG, la diversidad de temas que hemos tenido que abarcar, ha supuesto una tarea muy divertida, ya que siempre encontrábamos retos diferentes en el diseño o implementación.

11.4. TRABAJO DE FUTURO

Dado que las funcionalidades implementadas en la aplicación son muy básicas, lo más seguro es que no sigamos escalando esta aplicación, pero tenemos la posibilidad de rediseñarla desde cero, creemos que la parte más compleja sería encontrar la idea novedosa que supondría la motivación necesaria para comenzar el trabajo.

A parte de esto, los archivos de configuración que hemos implementado para Kubernetes e Istio, pueden servirnos en un futuro para cualquier aplicación, ya que cubren la mayor parte de las funcionalidades. Además, podríamos plantearnos configurar los *proxies* Envoy de los *pods* para darles las funcionalidades específicas de cada uno, y así ahorrarnos las configuraciones de reglas externas.

Por último, cabe señalar que también hemos obtenido conocimientos importantes en la elaboración de proyectos y seguimiento de metodologías, que nos pueden ayudar en nuestra vida laboral.

12. CONCLUSIONS

In this project we have implement an application of microservices with a service mesh, we have been focused on developing and understanding most of the features that Istio provides. It is true that the implementation of services is quite basic, but it is enough to reach all the objectives and learn deeply about Istio's configurations. Knowing this, we are going to expose our conclusions.

12.1. ADVANTAGES AND DISADVANTAGE

In this section we decided to make this part by comparing two general aspects that we know thanks to the implementation of this project. The first comparison and most obvious is the monolithic architecture and micro services.

Aspects	Monolithic	Micro Services
Modularity	It's a very stiff architecture and very hard to adapt, in other words, having all the applications and functionality in the same layer impedes its independent development.	It's based in autonomic services, with a vast diversity in size and functionality that can be developed independently.
Scalability	It can be hard because it has a low modularity and must scale vertically.	Unlike monolithic architecture, it can perform independent services more quickly and effective.
Versatility	Same technology.	We can introduce multiple languages and work with different environments at the same time.
Design	Unique and fast. Like it is in the same layer, the design takes less time.	It can be done in many ways, the most common one begins with a monolithic architecture and break it up in different services. Another one is starting from the bottom, which means a tedious work and a lot of time.
Implementation	Like the system is on the same layer the guidelines are clear, it can execute communications between participants, but it has to be in a specific order, and it can't be used until most of the development is done.	Like we said before, it can be done independently, although the communication between groups must be continuous, aiming to lead the application to the same destiny. Besides, it requires more specialized groups, this can be an advantage or not

		depending on how you look at it, since more specialized groups will allow the development of a les consuming application using less resources and less functionalities.
Cost	The implementation cost is lower.	The implementation cost is higher because it means a high cost in infrastructure.
Maintenance	It is usually costly, which means the application life cycle will be shorter.	Being a system with a high modularity, the maintenance can be done effectively, besides it is easier to control the failures than in monolithic architecture.
Production	Simple and widely known.	Simple, although you can detect errors can be detected due to lack of communication between developers.

Tabla 8-Monolithic and microservices architecture comparison

The next aspect that we should analyze is the use of a service mesh versus a developed application and controlled with a container orchestrator. As we have seen throughout this project, the service mesh, specifically Istio, covers much more functionality (canary releases, blue deployment, authentication, etc.) than an orchestrator, in addition it allows us to monitor various metrics. The main disadvantage of a service mesh is that it increases the complexity of our architecture, since we introduce elements like proxies, centralized control panel or iptables rules. Another disadvantage of Istio is that it increases the latency of the communications, although this can be acceptable, contrasting with the multiple functionalities that it offers.

12.2. GROUP OPINIONS

In this section, we try to transmit the sensations that we have been able to get after the elaboration of this project, specifically, those that prevail at the end of the work.

The main feeling we have is that once we have understand the microservices architecture and, in particular, the service mesh, we believe it is a powerful tool with a great future in application development. By being able to implement the most part of Istio functionality, we have understood the multiple applications that it can have.

At the beginning of the project, when we were still studying the subject, we found out the amount of companies that made the jump to micro services architecture (Netflix for example), or companies giving the first step towards this change. Now that we know enough to give an objective opinion, we understand why they have chosen this kind of architecture. It is true that most of the companies are big ones, with a huge amount of

resources and experts in the area. However, even if we do not have the resources, choosing this architecture is not going to be a mistake.

12.3. OBJECTIVES ACHIEVED

At the beginning of this project, we introduced the main and specific objectives and confronted them at first. The main objective was comprehending and implementing a microservice architecture using a service mesh.

We believe we have exposed enough information in this project to assert that the main objective has been accomplished. We faced this work as a challenge to understand an architecture that we had never meet before in any other course, and we finish it as a future option. The world of computer science is very broad and few specialties or subjects have the beauty of encompassing the knowledge of two areas as specific as application development and computer architecture, that is why maybe in the future we would like to investigate more about the knowledge acquired.

About the specific objectives, we have accomplished the expectations of implementing an application with multiple languages and platforms. Thanks to the courses we have taken, we just needed to remember some guidelines to use Java, Python, SQL or PHP, but we had to start from scratch and delve into Spring Boot and the Flask version of Python.

In summary, we are very happy with the decision we made choosing this project. The diversity of topics that we have had to cover has been a very fun task, since we always encountered different challenges in the design or implementation.

12.4. FUTURE WORK

Given that the functionality implemented in the application is very basic, it's safe to say we will not deepen in this application, but we have the possibility to residing it from scratch, we think that the most difficult part would be finding an original idea to motivate the start of this work.

Besides this, we can use the configuration archives that we have implemented for Kubernetes and Istio in any other application in the future, since they cover most of the functionality. Also, we could set up the proxies from Envoy of the pods so we can give them the specific functionalities to each one, and thus save the setting up of the external rules.

Finally, it should be noted that we have also obtained important knowledge in the development of projects and monitoring of methodologies, which can help us in our working life.

13. APORTACIÓN DE LOS PARTICIPANTES

En este apartado vamos a exponer lo que hemos aportado cada participante al desarrollo del trabajo. Vamos a comenzar explicando cómo nos hemos dividido el trabajo y el funcionamiento general del equipo, y posteriormente, lo analizaremos individualmente.

Este apartado está muy relacionado con el de la planificación ya que vamos a hablar de las diferentes tareas que hemos realizado.

El primer paso del TFG, el estudio y aprendizaje del tema, lo realizamos individualmente, pero sobre la misma información. Es decir, ya que los dos participantes obtuvimos una beca en la plataforma de enseñanza online OpenWebinars, realizamos los mismos cursos. Si es verdad, que la investigación de artículos, blogs y portales Web se realizó independientemente, pero cada vez que alguno de los participantes encontraba información de interés lo compartía con el otro participante.

El diseño de la arquitectura fue realizado en las instalaciones de la Facultad de Informática, y con los dos participantes presentes, ya que se realizó antes del confinamiento.

La implementación de la arquitectura de microservicios se realizó individualmente, siguiendo la siguiente división:

- Front: Ricardo Daniel Cabrera Lozada.
- Python (versión 1 y versión 2): Raúl Diego Navarro.
- Usuario: Ricardo Daniel Cabrera Lozada y Raúl Diego Navarro.
- Empleado (versión 1 y versión 2): Ricardo Daniel Cabrera Lozada y Raúl Diego Navarro.
- SQL: Raúl Diego Navarro.

En los servicios que están implementados por los dos participantes, se han dividido las clases que forman el servicio, con el objetivo de ir más rápido.

La implementación necesaria de Docker se basa en el correcto desarrollo de los Dockerfiles, además de una lista de comandos necesarios para su rápida utilización. La implementación de los Dockerfiles se ha llevado a cabo de la siguiente manera:

- El primer Dockerfile fue el del servicio “Usuario” y se realizó en conjunto por los dos participantes, ya que creímos necesario asegurarnos de que el primer paso en esta plataforma se realizaba correctamente.
- Los demás Dockerfiles se dividen en:
  - Front: Ricardo Daniel Cabrera Lozada.
  - Empleado: Ricardo Daniel Cabrera Lozada.
  - Empleado (versión 2): Raúl Diego Navarro.
  - Python (versión 1 y versión 2): Raúl Diego Navarro.

El despliegue en Docker se hizo en presencia de los dos participantes, y siguiendo los pasos expuestos en el apartado de implementación.

En Kubernetes e Istio fue algo distinto, ya que la mayoría de los archivos eran desarrollados por un participante y complementados o corregidos por el otro participante.

Como expusimos en el apartado de Kubernetes el primer paso fue desplegar un *pod* y esto se realizó nuevamente en presencia de los dos participantes. A partir de aquí, nos dividimos los archivos al 50%, es decir, cada uno hizo la mitad de los *deployments* y los *services* respectivos, excepto en el caso de la base de datos que fue Ricardo Daniel Cabrera Lozada el que realizó el *deployment* y el *service*, y Raúl Diego Navarro el que desarrollo los *volumes* y el *secret*.

En este caso el despliegue fue realizado individualmente por cada uno de los participantes. Decidimos que fuera así para aprender de forma independiente los errores e inconvenientes que puede conllevar el despliegue de una demo como esta.

En Istio, la inyección del *proxy* Envoy fue realizada por cada uno de los participantes sobre sus respectivos archivos de Kubernetes. De la misma manera se llevó a cabo el despliegue de la demo en Istio.

La implementación de las *DestinationRules* fue desarrollada por los dos participantes del grupo, ya que era el paso inicial, y decidimos hacerlo en conjunto, dada su importancia. En el caso de los *VirtualServices* se ha dividido de la siguiente manera:

- Blue Green Deployment (3 archivos): Han sido desarrollados por Ricardo Daniel Cabrera Lozada.
- Canary Releases (2 archivos): Han sido desarrollados por Raúl Diego Navarro.
- Dark launches (2 archivos): Han sido desarrollados por Raúl Diego Navarro.
- Load Balancer (1 archivo): Ha sido desarrollado por Ricardo Daniel Cabrera Lozada.

Hay que recordar que sobre todo para esta parte, todos los archivos han sido revisados por los dos participantes y comprobados en ejecución, tanto los *DestinationRules* como los *VirtualServices*. En el caso de los archivos de configuración de Circuit Breaker, Pool Ejection y cifrado TLS fueron desarrollados por los dos participantes en conjunto, a través de una plataforma de videoconferencia.

Por último, la memoria ha sido estructurada y diseñada en conjunto. Hemos divido la memoria de la siguiente manera:

- El primer participante ha elaborado el primer punto o subpunto (dependiendo de la longitud y dificultad).
- El segundo participante revisa y corrige el punto elaborado por su compañero, y realiza el siguiente apartado
- Este procedimiento se ha mantenido durante toda la memoria, pero no podemos establecer exactamente qué orden hemos seguido, ya que hemos cambiado la estructura de la memoria durante la elaboración de esta.

Pasamos a la exposición individual de cada participante.

RAÚL DIEGO NAVARRO

En toda mi vida universitaria nunca había participado en un equipo en el que el trabajo haya estado tan igualado, creo firmemente que la autoría del proyecto en su totalidad es del 50%. Al haber trabajado anteriormente en otros proyectos con Ricardo hemos podido compaginarnos perfectamente, una de las cosas que he notado a lo largo del trabajo ha sido la buena respuesta ante las adversidades que nos encontrábamos en el camino.

Respecto a mi parte de implementación debo exponer varias cuestiones. La primera es que la implementación de los servicios en Python la he llevado a cabo, ya que he cursado la asignatura de Aprendizaje Automático y Big Data y en ella he aprendido a programar en Python. La implementación de la base de datos ha sido sencilla y por tanto no tengo nada que añadir.

En Docker, una vez que habíamos desarrollado el primer Dockerfile, todo ha ido “sobre ruedas”, debido a que eran archivos parecidos entre sí y que los cambios eran puntuales, como vimos en el apartado de implementación.

En el caso de Kubernetes, creo que la división ha sido la mejor, ya que nos ha permitido entender y conocer cualquier archivo y campo, aunque no haya sido desarrollado por los dos. También cabe señalar que al hacer el despliegue individual conocemos la totalidad de los elementos que forman la arquitectura.

En Istio, una de las partes más importantes del proyecto, hemos conseguido una metodología adecuada que nos permitiera explicar cada archivo en la memoria sin importar su autor.

Por último, la memoria como hemos expuesto antes ha sido elaborada por los dos participantes en conjunto, y me gustaría señalar la importancia que hemos tenido cada uno en todas las secciones, puesto que hemos trabajado los dos sobre todas ellas.

RICARDO DANIEL CABRERA LOZADA

En la realización de este proyecto el trabajo realizado fue bien distribuido, ya que ambos hemos trabajado por igual. Nos hemos ayudado en todos los problemas que han surgido en el trascurso del proyecto.

El apartado de la página de inicio la he realizado en PHP, debido a que ya tenía conocimientos previos del lenguaje y se me ha hecho más fácil la implementación de este. Así mismo, en el trascurso de la carrera he realizado la asignatura Aplicaciones Web, que me aportó todos estos conocimientos.

Con respecto a Docker se ha realizado distintos Dockerfiles y configuraciones para poder desplegar la aplicación. No fue muy complejo ya que hemos seguido la documentación oficial.

En el momento de comenzar con Kubernetes, ambos hemos participado en las distintas implementaciones al igual que en Docker. Se desarrollo los distintos archivos de configuración los cuales permite desplegar la aplicación en un clúster, para poder probar todos los beneficios que nos da el mismo.

En Istio, se desarrolló de igual manera archivos de configuración y además se configuró todo el entorno para el correcto funcionamiento del proyecto.

La memoria se ha sido realizado por ambos participantes de manera equitativa. Cabe destacar que todo el proyecto se ha divido entre los dos y hemos intervenido en cada proceso para llegar al objetivo.

14. REFERENCIAS

14.1. REFERENCIAS BIBLIOGRÁFICAS

1. Newman S. (2015). *Building Microservices*. EEUU: O'Reilly.
2. Wolff E. (2016). *Flexible Software Arquitecture*. Germany.
3. Macero M. (2017). *Learn Microservices with Spring Boot*. EEUU: Apress.
4. Nadareishvili I, Mitra R, McLarty M. & Admundsen M. (Coord.) (2016). *Microservice Architecture*. EEUU: O'Reilly.
5. Posta C. & Sutter B. (Coord.) (2019*). Introducing Istio Service Mesh for Microservices.* EEUU: O'Reilly.
6. Lioy J. (2019). *Docker: The complete guide to the most widely used virtualization technology. Create containers and deploy them to production safely and securely.*
7. Stoneman E. (2020). *Learn Docker in a Month of Lunches*. EEUU: Manning.
8. Arnold Z., Dua S., Huang W., Masood F., Qin M. & Abu Taleb M. (Coord.) (2020). *The Kubernetes Workshop: A New, Interactive Approach to Learning Kubernetes*. Gran Bretaña-India: Packt Publishing Limited.
9. Dobies J. (2020). *Kubernetes Operators: Automating the Container Orchestration Platform*. Gran Bretaña: O'Reilly.
10. Calcote L. & Butcher Zack (Coord.) (2019). *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. EEUU: O'Reilly.

14.2. ARTICULOS DE INTERÉS

1. Soto A. (2019). *Diferencia entre arquitectura monolítica y de microservicios.* Dirección web: https://openwebinars.net/blog/diferencia-entre-arquitectura-monolitica-y-microservicios/.
2. Balalaie A., Heydarnoori A. & Jamshidi P. (Coord.) (2016). *Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture*. Dirección web: https://ieeexplore.ieee.org/abstract/document/7436659
3. Desconocido. (2019). *Arquitectura de microservicios*. Dirección web: https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices.
4. Morales C. (2019*). Comparativa de orquestadores: Docker Swarm vs Kubernetes vs Apache Mesos*. Dirección web: https://profile.es/blog/comparativa-de-orquestadores-docker-swarm-vs-kubernetes-vs-apache-mesos/
5. Red Hat (2019). *ARQUITECTURA DE MICROSERVICIOS ¿Qué es Service Mesh?* Dirección web: https://www.redhat.com/es/topics/microservices/what-is-a-service-mesh
6. Smith F. (2018). *What Is a Service Mesh?* Dirección web: https://www.nginx.com/blog/what-is-a-service-mesh/
7. Mora L. (2019). *Probando Linkerd, el pionero de las services mesh.* Dirección web: https://www.paradigmadigital.com/dev/probando-linkerd-el-pionero-de-los-services-mesh/

8. IBM Cloud (2020). *Conceptos de la plataforma de nube*. Dirección web: https://cloud.ibm.com/docs/cloud-native?topic=cloud-native-platform&locale=es

14.3. ENLACES DE INTERÉS

1. Página web oficial Kubernetes. Dirección web: https://kubernetes.io/
2. Página web oficial Istio. Dirección web: https://istio.io/
3. Información general de la arquitectura de microservicios. Dirección web: https://aws.amazon.com/es/microservices/
4. Información general de contenedores. Dirección web: https://www.redhat.com/es/topics/containers
5. Información general de microservicios. Dirección web: https://www.redhat.com/es/topics/microservices
6. Arquitectura de microservicios. Dirección web: https://www.paradigmadigital.com/dev/consolida-arquitectura-microservicios-service-mesh/