



Istio

Nombre: Diana Fernanda Castillo Rebolledo

Materia: Computación Tolerante a Fallas

Horario: L-I 11:00 am-1:00 pm

Profesor: Dr. Michel Emanuel López Franco

Sección: D06

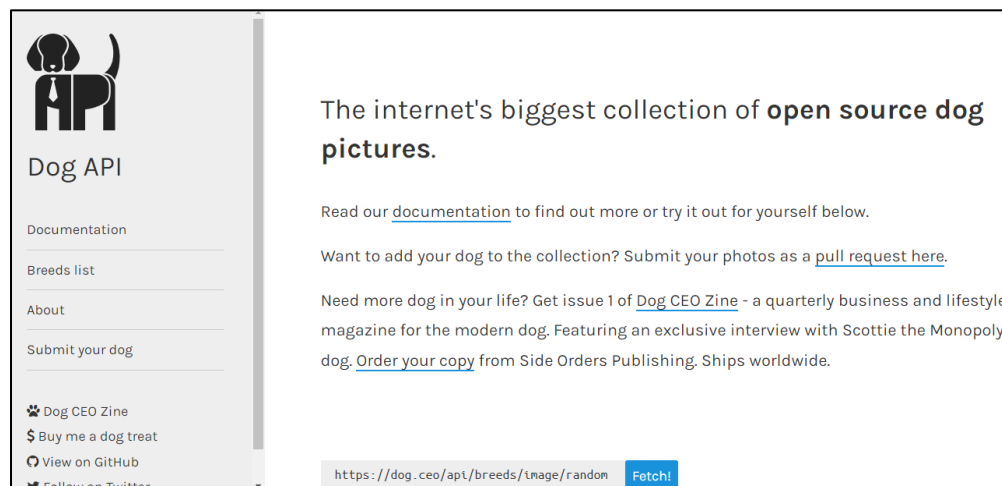
¿Qué es Istio?

Istio es una herramienta de código abierto que implementa un service mesh, esto es una arquitectura para facilitar la comunicación entre servicios o microservicios mediante un proxy, y así hacer las comunicaciones más seguras, rápidas y confiables, además proporciona gestión de tráfico, aplicación de políticas de cumplimiento y recolección de métricas. Esta plataforma puede ejecutarse en distintos entornos como on-premise, alojados en la nube, contenedores de Kubernetes y en servicios que se ejecutan en máquinas virtuales, entre otros.

La arquitectura de Istio se divide en el plano de datos y el plano de control. En el plano de datos, el soporte de esta plataforma se agrega a un servicio mediante la implementación de un proxy basado en “Envoy”, que es añadido a los pods como container “sidecar” dentro de su entorno. Este proxy se encuentra junto a un microservicio y envía las solicitudes desde y hacia otros proxies. En conjunto, dichos proxies forman una red que intercepta la comunicación de red entre los microservicios. El plano de control gestiona y configura los proxies para enrutar el tráfico. Este plano también configura los elementos para aplicar las políticas y recopilar datos de telemetría.

Una vez definido lo anterior, se explicará un ejemplo utilizando la herramienta de Istio para la gestión de un cluster de Kubernetes de una aplicación en Python.

Primero cree el programa de Python que consiste en una pequeña aplicación que obtiene datos a través de una API de perros llamada “Dog API”, la cual almacena links de imágenes de diversas razas:



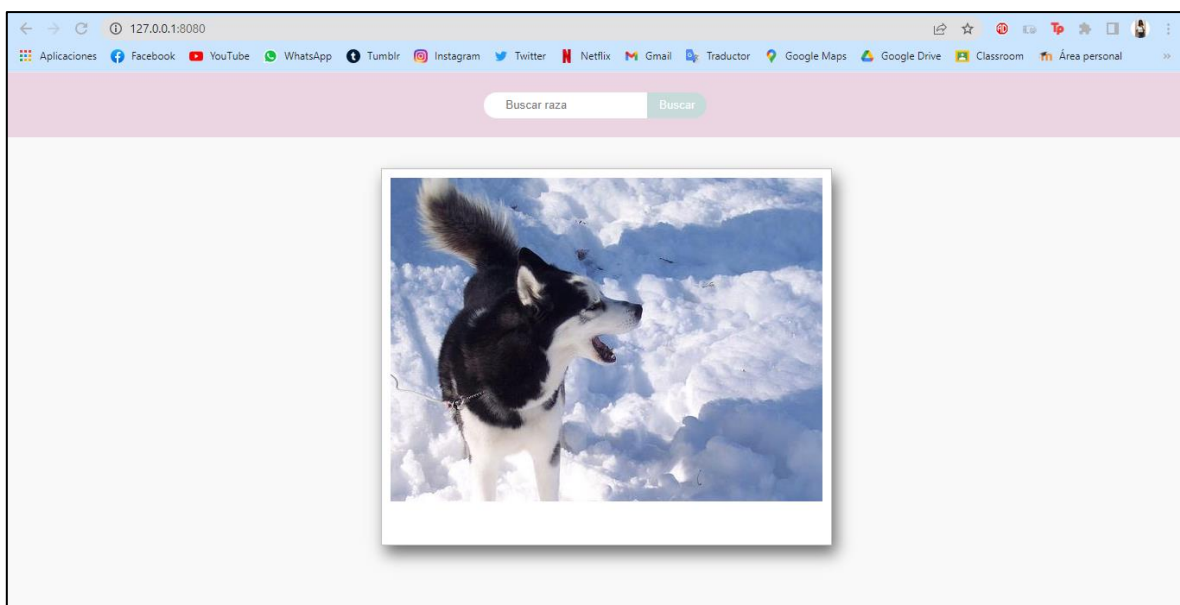
En el buscador de mi aplicación se puede buscar una raza y aparecerán imágenes random de perros de esa raza cada que se recargue la página, y así hasta que se cambie de raza en el buscador. Esto lo hice utilizando Flask como se muestra a continuación:

```

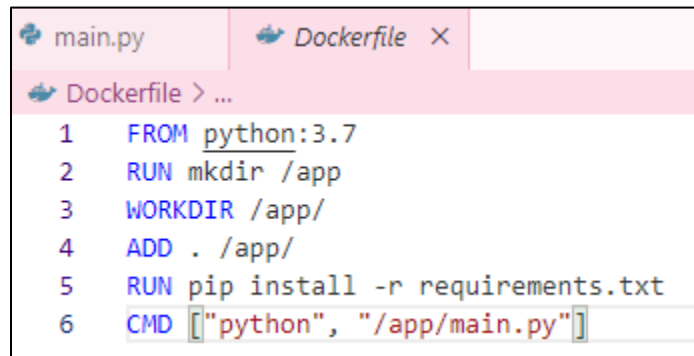
main.py > ...
1  from flask import Flask, render_template, request
2  import requests
3
4  app = Flask(__name__)
5
6  @app.route('/', methods=["GET", "POST"])
7  def index():
8      if request.method == 'GET':
9          url = "https://dog.ceo/api/breeds/image/random"
10
11         if request.method == 'POST':
12             name = request.form['name']
13             url = "https://dog.ceo/api/breed/{}/images/random".format(name)
14
15         resultado = requests.get(url)
16         datos = resultado.json()
17         imagen = datos["message"]
18
19         return render_template("index.html", imagen=imagen)
20
21
22 app.run(host="0.0.0.0", port=8080)

```

Para ello primero importé la librería de Flask e instancié un objeto Flask con el nombre de app, después con `@app.route('/')` definí el path para visualizar la aplicación que será la raíz, y dentro de una función indiqué el link de la API al que tenía que ir dependiendo si el request era un GET (imágenes random de cualquier raza) o POST (imágenes random de una raza en específico), después del request se guarda el resultado en un objeto json para después guardar solamente el valor que contiene la clave "message" que es el url de la imagen a mostrar. La función retorna un HTML que es la página donde interactúa el usuario y se muestran las imágenes. Fuera de la función con `app.run()` se define que se ejecutará el objeto app de Flask, en el puerto 8080. Al ejecutar la app se muestra de esta manera:



Una vez implementada la aplicación, la metí en un contenedor de Docker para posteriormente utilizar Kubernetes e Istio, entonces primero creé el archivo Dockerfile:



```
main.py Dockerfile X
Dockerfile > ...
1 FROM python:3.7
2 RUN mkdir /app
3 WORKDIR /app/
4 ADD . /app/
5 RUN pip install -r requirements.txt
6 CMD ["python", "/app/main.py"]
```

En requirements.txt coloqué flask y requests porque son los módulos que se necesitan instalar para que la aplicación de Python funcione.

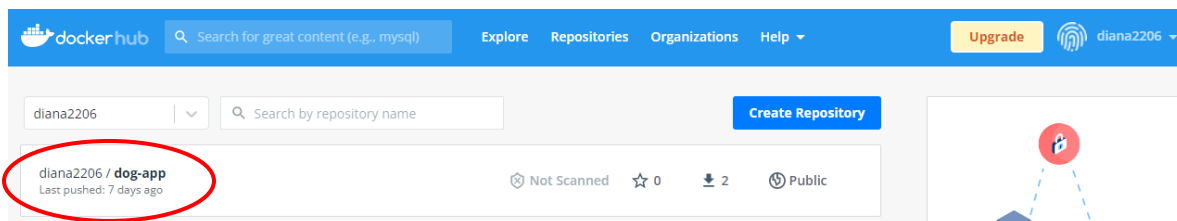
En seguida construí la imagen (a la cual le llamé “dog-app”) con el comando:

```
docker build -t dog-app .
```

Una vez que se construye la imagen, la subí a Docker Hub para utilizarla posteriormente, para subirla utilicé los siguientes comandos:

```
docker tag dog-app diana2206/dog-app:v1
```

```
docker push diana2206/dog-app:v1
```



Coloqué el nombre de esta imagen en el deployment.yaml que utilicé para implementar la aplicación en Kubernetes. Este archivo se compone de dos partes como en la actividad anterior, una es para el servicio LoadBalancer y la otra es de la aplicación, se define que tendrá 2 réplicas o pods. En la parte inferior se observa que este archivo está conectado a la imagen de Docker que subí a Docker Hub que es de la aplicación, por lo que para implementar la aplicación en el clúster de Kubernetes se estará usando esa imagen:

```

! deployment.yaml
2  kind: Service
3  metadata:
4    name: flask-service
5  spec:
6    selector:
7      app: flask-app
8    ports:
9      - protocol: "TCP"
10      port: 8000
11      targetPort: 8080
12    type: LoadBalancer
13
14  ---
15  apiVersion: apps/v1
16  kind: Deployment
17  metadata:
18    name: flask-app
19  spec:
20    selector:
21      matchLabels:
22        app: flask-app
23    replicas: 2
24    template:
25      metadata:
26        labels:
27          app: flask-app
28      spec:
29        containers:
30          - name: flask-app
31            image: docker.io/diana2206/dog-app:v1
32            imagePullPolicy: IfNotPresent
33            ports:
34              - containerPort: 8080

```

Para implementar Kubernetes utilicé minikube, iniciándolo con el siguiente comando:

minikube start --driver=virtualbox --no-vtx-check

Entonces una vez iniciado, ejecuté el archivo yaml en kubernetes con el siguiente comando:

kubectl apply -f deployment.yaml

Una vez hecho lo anterior, se crean los pods y el LoadBalancer en el cluster:

```
Administrador: Windows PowerShell
PS C:\WINDOWS\system32> kubectl get all
NAME                                READY    STATUS    RESTARTS    AGE
pod/flask-app-579669fbd4-dx15p      1/1      Running   3 (96m ago)  19h
pod/flask-app-579669fbd4-qszk6      1/1      Running   3 (96m ago)  19h

NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
service/flask-service               LoadBalancer  10.102.124.70 <pending>      6000:32602/TCP   19h
service/kubernetes                   ClusterIP      10.96.0.1     <none>         443/TCP          21h

NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
deployment.apps/flask-app            2/2      2              2            19h

NAME                                DESIRED    CURRENT    READY    AGE
replicaset.apps/flask-app-579669fbd4 2          2          2        19h
PS C:\WINDOWS\system32>
```

La aplicación se va a mostrar a través del LoadBalancer, pero para eso se necesita que se genere una IP externa, y para ello se ejecuta el comando minikube tunnel.

Después de todo esto, para utilizar Istio primero se instaló Istioctl en el cluster de Kubernetes con el siguiente comando:

istioctl install --set profile=demo -y

Después se tienen que configurar todos pods del namespace para que se les inyecte el proxy de Istio, para hacer esto se tiene que poner una etiqueta al namespace istio injection de la siguiente manera:

kubectl label namespace default istio injection=enabled

A partir de esto cualquier pod que se despliegue en este namespace se le inyectará el proxy de Istio, para esto eliminé los pods y los volví a crear, para que ya tengan dos contenedores en vez de uno:

NAME	READY	STATUS	RESTARTS	AGE
flask-app-86dc77d857-8fths	2/2	Running	0	81s
flask-app-86dc77d857-q689n	2/2	Running	0	81s

Para poder verificar los dos contenedores creados en cada pod, se puede describir uno de los pods de esta manera:

Kubectl describe pod flask-app-86dc77d857-8fths

En la parte de containers se tienen dos contenedores, uno es el de flask-app que es el de la aplicación, y además Istio inyectó el otro que es del proxy llamado istio-proxy, este es el contenedor que se va a encargar de monitorizar todo lo que ocurra en este pod, como las peticiones que este reciba.

```
flask-app:
  Container ID:   docker://48b223bc80a8cca7ee8af2a841a5f01336f25d8dcb055f799d4438b1d480da05
  Image:          docker.io/diana2206/dog-app:v1
  Image ID:       docker-pullable://diana2206/dog-app@sha256:134f979cc8db82bd3eced36ecfbfa51c5818db1afa810898016884dd6895d02d
  Port:          5000/TCP
  Host Port:     0/TCP
  State:         Running
  Started:       Fri, 22 Apr 2022 16:51:49 -0500
  Ready:         True
  Restart Count: 0
  Environment:   <none>
  Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-rbxwf (ro)
```

```
istio-proxy:
  Container ID:   docker://6edcf61886747763cbbec67bacfdb0342b51d651f5ae6ca63b78d411efb4ea35
  Image:          docker.io/istio/proxyv2:1.12.6
  Image ID:       docker-pullable://istio/proxyv2@sha256:4b796185f4eecb8bc408bcb3c0f74f1e2065d06992d6430a32e12e6b9767aad8
  Port:          15090/TCP
  Host Port:     0/TCP
  Args:
    proxy
    sidecar
    --domain
    $(POD_NAMESPACE).svc.cluster.local
    --proxyLogLevel=warning
    --proxyComponentLogLevel=misc:error
    --log_output_level=default:info
    --concurrency
    2
  State:         Running
  Started:       Fri, 22 Apr 2022 16:51:55 -0500
  Ready:         True
  Restart Count: 0
  Limits:
    cpu:         2
    memory:      1Gi
  Requests:
    cpu:         10m
    memory:      40Mi
  Readiness:     http-get http://:15021/healthz/ready delay=1s timeout=3s period=2s #success=1 #failure=30
```

Para poder observar más visual y gráficamente todo lo que está pasando y la comunicación en Istio, está la herramienta de Kiali, así como Prometheus para recopilar métricas de memoria, CPU y disco de los microservicios, y Grafana para visualizar las métricas que manda Prometheus. Para instalarlas se ejecutan los siguientes comandos:

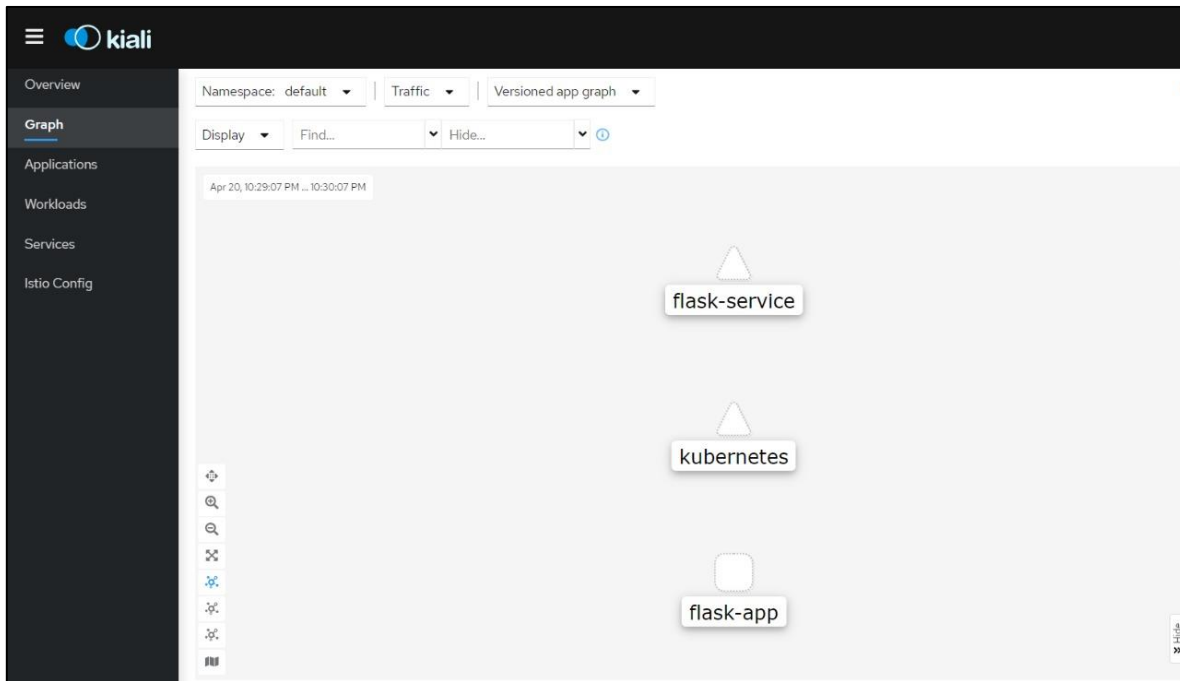
```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.13/samples/addons/kiali.yaml
```

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.13/samples/addons/grafana.yaml
```

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.13/samples/addons/prometheus.yaml
```

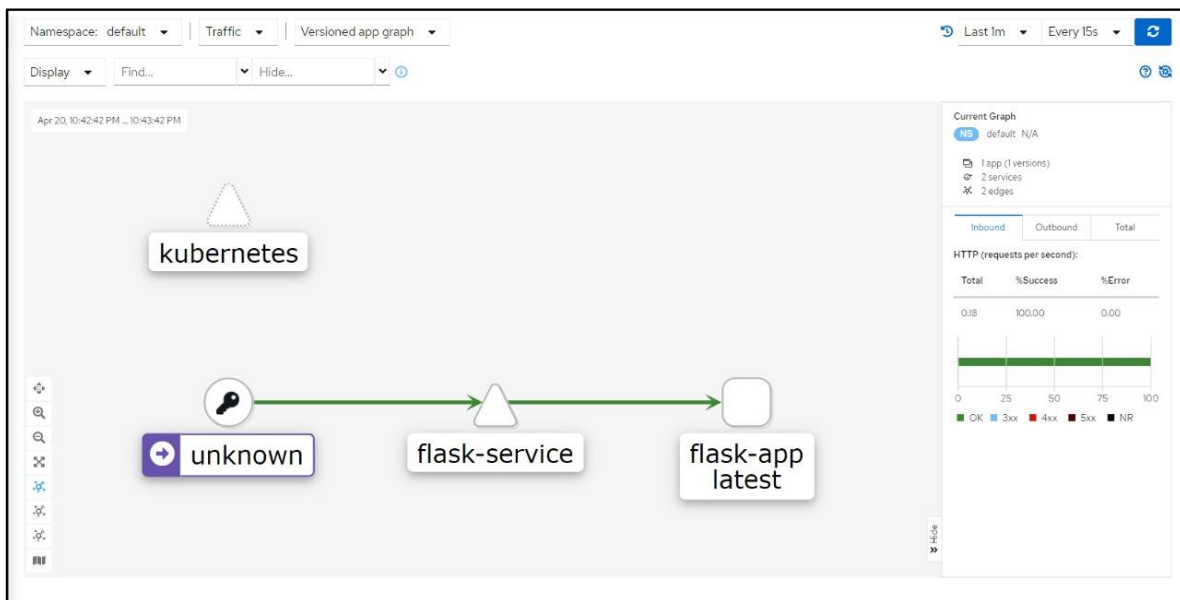
Ahora se va a levantar la interfaz de Kiali, la cual muestra un dashboard en el navegador, para ello se utiliza el comando:

```
istioctl dashboard kiali
```



En Kiali aparece la aplicación, los triángulos son los servicios y el cuadrado es de los pods.

Al estar haciendo un request cada segundo a la aplicación, al actualizar la página en Kiali se comienza a ver el tráfico:

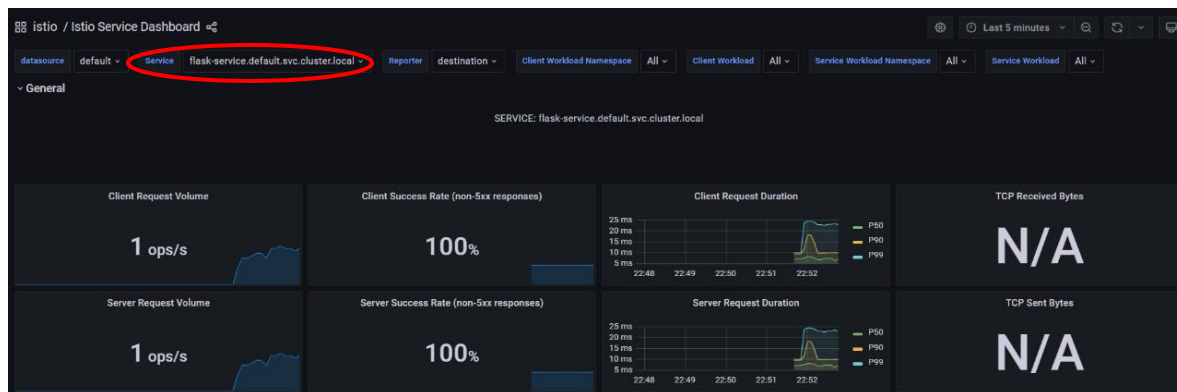


Unknown es de donde provienen los request a la aplicación ya que vienen desde afuera, en este caso es de un pequeño script que realiza requests cada segundo a la app, y esas peticiones pasan a través del LoadBalancer que es flask-service, y a su vez este accede a los pods de la aplicación, y así va funcionando el tráfico. Se aprecia que está de un color verde y según la simbología de la derecha eso indica que todo está bien.

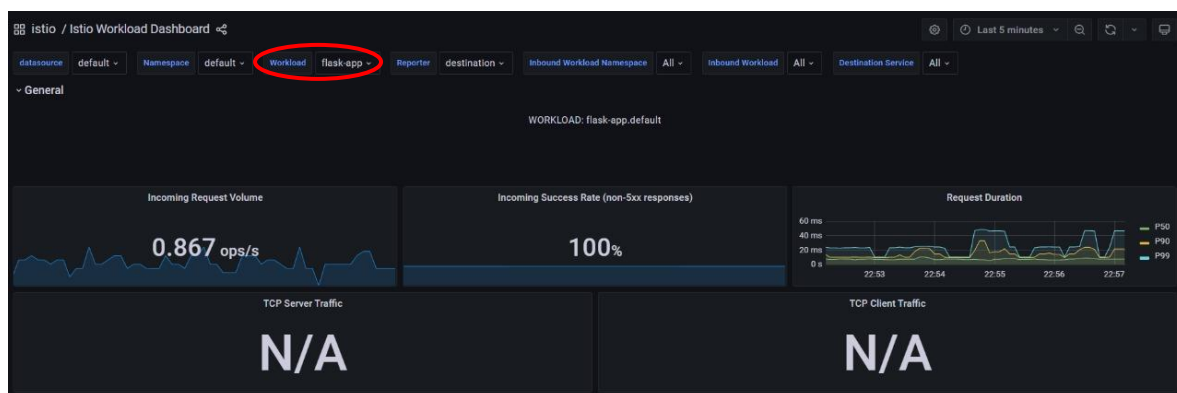
En Grafana se pueden observar las métricas que se obtienen desde Prometheus, a Grafana se accede de la siguiente manera:

istioctl dashboard grafana

Aquí se observan las estadísticas del tráfico del servicio, indica que se hace un request cada segundo y que todas las peticiones se realizan correctamente.



También se puede observar el tráfico y carga de trabajo del pod de la app:



Código en GitHub: <https://github.com/diana-castillo/Istio.git>

Imagen en Docker Hub:

<https://hub.docker.com/repository/docker/diana2206/dog-app>

Conclusiones:

Esta actividad me pareció bastante interesante, pues no pensé que hubiera más herramientas por encima de Kubernetes. El service mesh me parece una arquitectura muy organizada para las aplicaciones grandes y con funciones más críticas, pues así se dividen las cargas y funciones en microservicios, y esto ayuda a poder darle un mejor mantenimiento. Istio es una herramienta de service mesh muy completa, ya que en ella se puede observar y gestionar el estado de toda la arquitectura de una aplicación para así

encontrar fácilmente si es que hay errores o incongruencias en los servicios, además de que de esta manera se puede observar todo lo que sucede, como el tráfico que pasa, o también es muy útil cuando se tienen varias versiones de una misma funcionalidad pero aun no se quiere colocar la nueva versión al cien por ciento en producción por si es que tiene fallos, se puede colocar pero dividir la carga de tráfico entre esta y la antigua versión poco a poco para asegurarse de que realmente funcione.

Referencias:

Casallas, R. [AprenDevOps]. (2021, 25 noviembre). *Introducción a ISTIO y SERVICE MESH - Explicación en 10 minutos* [Video]. YouTube. <https://www.youtube.com/watch?v=z4eqOiDIFoE>

Casallas, R. [AprenDevOps]. (2021b, diciembre 9). *Instalación y Configuración de ISTIO en KUBERNETES / Explicación sencilla* [Video]. YouTube. <https://www.youtube.com/watch?v=nRxWnBE0wzE>

Pelado Nerd. (2021, 16 febrero). *Introducción a ISTIO / Service Mesh* [Video]. YouTube. <https://www.youtube.com/watch?v=ofJ5swfP2kQ>

TechWorld with Nana. (2021, 1 enero). *Istio & Service Mesh - simply explained in 15 mins* [Video]. YouTube. <https://www.youtube.com/watch?v=16fgzklcF7Y>

¿Qué es Istio? (2019, 8 enero). Red Hat. Recuperado 21 de abril de 2022, de <https://www.redhat.com/es/topics/microservices/what-is-istio>

Qué es Istio kubernetes y cómo simplificar el cluster. (s. f.). ACK Storm. Recuperado 21 de abril de 2022, de <https://www.ackstorm.com/que-es-istio-kubernetes-como-simplificar-cluster/>