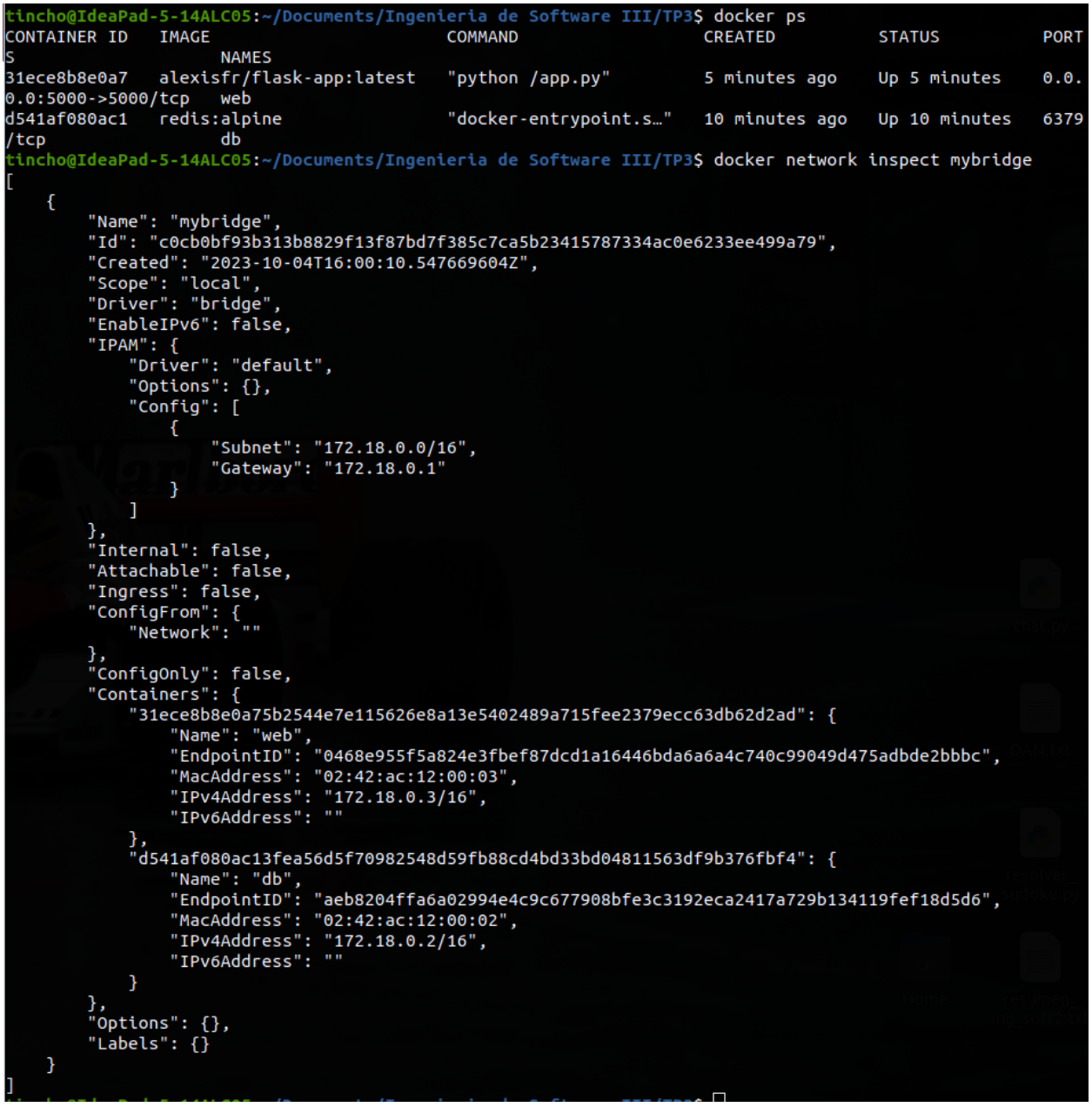
Trabajo Práctico 3 - Arquitectura de Sistemas Distribuidos

1- Sistema distribuido simple

```
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$ docker network create -d bridge mybr
idge
c0cb0bf93b313b8829f13f87bd7f385c7ca5b23415787334ac0e6233ee499a79
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$  docker run -d --net mybridge --name
 db redis:alpine
Unable to find image 'redis:alpine' locally
alpine: Pulling from library/redis
96526aa774ef: Pull complete
6adfacd3b74c: Pull complete
8f2d8ff49f68: Pull complete
473eef84775a: Pull complete
a8fc03039a58: Pull complete
4f4fb700ef54: Pull complete
3c27b2421cc2: Pull complete
Digest: sha256:343e6546f35877801de0b8580274a5e3a8e8464cabe545a2dd9f3c78df77542a
Status: Downloaded newer image for redis:alpine
d541af080ac13fea56d5f70982548d59fb88cd4bd33bd04811563df9b376fbf4
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$   docker run -d --net mybridge -e RE
DIS_HOST=db -e REDIS_PORT=6379 -p 5000:5000 --name web alexisfr/flask-app:latest
Unable to find image 'alexisfr/flask-app:latest' locally
latest: Pulling from alexisfr/flask-app
f49cf87b52c1: Pull complete
7b491c575b06: Pull complete
b313b08bab3b: Pull complete
51d6678c3f0e: Pull complete
09f35bd58db2: Pull complete
1bda3d37eead: Pull complete
9f47966d4de2: Pull complete
9fd775bfe531: Pull complete
2446eec18066: Pull complete
b98b851b2dad: Pull complete
e119cb75d84f: Pull complete
Digest: sha256:250221bea53e4e8f99a7ce79023c978ba0df69bdfe620401756da46e34b7c80b
Status: Downloaded newer image for alexisfr/flask-app:latest
31ece8b8e0a75b2544e7e115626e8a13e5402489a715fee2379ecc63db62d2ad
```

Hello from Redis! I have been seen 1 times.



Primero, ejecuté docker ps para verificar los contenedores en ejecución y los puertos abiertos. Los puertos abiertos en los contenedores son:

- El contenedor "web" tiene el puerto 5000 mapeado al puerto 5000 del host.
- El contenedor "db" no tiene puertos explícitamente mapeados.

Luego, utilicé el comando docker network inspect mybridge para mostrar detalles de la red mybridge en Docker. Aquí pude ver información detallada sobre la configuración de la red, las subredes, las direcciones IP asignadas a los contenedores, y otros detalles relacionados con la red.

2- Análisis del sistema

```
# Importar las bibliotecas necesarias
import os
from flask import Flask
from redis import Redis

# Crear una instancia de la aplicación Flask
app = Flask(__name__)

# Establecer una conexión con el servidor de Redis utilizando las variables de entorno
# REDIS_HOST y REDIS_PORT para obtener la dirección IP y el puerto del servidor Redis.
# Las variables de entorno se establecerán cuando se ejecute el contenedor.
redis = Redis(host=os.environ['REDIS_HOST'], port=os.environ['REDIS_PORT'])

# Obtener el puerto de enlace del contenedor desde la variable de entorno BIND_PORT
bind_port = int(os.environ['BIND_PORT'])

# Definir una ruta para la página principal ("/")
@app.route('/')
def hello():
    # Incrementar el contador de visitas en Redis
    redis.incr('hits')

    # Obtener el valor actual del contador de visitas desde Redis y convertirlo a cadena
    total_hits = redis.get('hits').decode()

    # Devolver un mensaje que muestra cuántas veces ha sido visitada la página
    return f'Hello from Redis! I have been seen {total_hits} times.'

# Comprobar si este script se está ejecutando como el programa principal
if __name__ == "__main__":
    # Iniciar la aplicación Flask para escuchar en todas las direcciones IP ("0.0.0.0"),
    # habilitar el modo de depuración y utilizar el puerto especificado en BIND_PORT.
    app.run(host="0.0.0.0", debug=True, port=bind_port)
```

- Respecto a los parámetros -e en el segundo docker run del ejercicio 1, se utilizan para establecer variables de entorno en el contenedor. En este caso, se utilizan para configurar la dirección IP y el puerto de la base de datos Redis a la que se conectará la aplicación Flask. Estas variables de entorno (REDIS_HOST y REDIS_PORT) se utilizan en el código de la aplicación para conectarse correctamente a Redis.

- Si ejecutamos docker rm -f web y luego volvemos a correr docker run -d --net mybridge -e REDIS_HOST=db -e REDIS_PORT=6379 -p 5000:5000 --name web alexisfr/flask-app:latest, lo que sucederá es que se detendrá y eliminará el contenedor llamado "web" y se creará uno nuevo con la misma configuración. La aplicación web

comenzará desde cero en cuanto al contador de visitas, ya que Redis almacena este contador en la base de datos y no en el contenedor de la aplicación web.

- Si borramos el contenedor de Redis con docker rm -f db, la página web ya no podrá conectarse a Redis para contar las visitas, lo que resultará en un mensaje de error o una respuesta vacía en la página web. Sin embargo, si levantamos nuevamente un contenedor de Redis con docker run -d --net mybridge --name db redis:alpine, la aplicación web podrá conectarse nuevamente a Redis y seguir contando las visitas desde cero.

- Para no perder la cuenta de las visitas, se necesitaría una solución de almacenamiento persistente para el contador, como una base de datos SQL o un almacenamiento de claves-valor duradero como Redis. Esto permitiría mantener el contador de visitas incluso si los contenedores se detienen o recrean.

3- Utilizando docker compose

```
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$ docker-compose up -d
Creating network "tp3_default" with the default driver
Creating volume "tp3_redis_data" with default driver
Creating tp3_db_1 ... done
Creating tp3_app_1 ... done
```

Hello from Redis! I have been seen 1 times.

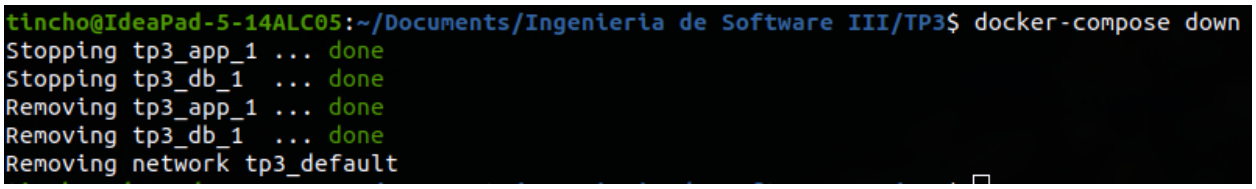
```
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$ docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED              STATUS              PORTS
               NAMES
b0990517ab0b   alexisfr/flask-app:latest   "python /app.py"         About a minute ago   Up About a minute   0.0.0.0:50
00->5000/tcp   tp3_app_1
6a92b8083bc4   redis:alpine               "docker-entrypoint.s…"   About a minute ago   Up About a minute   6379/tcp
               tp3_db_1
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$ docker network ls
NETWORK ID     NAME          DRIVER    SCOPE
11d57a54c70b   bridge        bridge    local
c9162e24098d   host          host      local
2c73b4e99ae8   none          null      local
d996e86a0773   tp3_default   bridge    local
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$ docker volume ls
DRIVER    VOLUME NAME
local     05a29f569d3cd951e297f56b54905f406043c2a61ea7496198deaae39d723c20
local     a2adaa00906606c5db571d521a2da64711707ed53fbac542488da4d7176490ba
local     tp3_redis_data
```

Docker Compose creó automáticamente los contenedores necesarios para mis servicios. En este caso, definí dos servicios en el archivo docker-compose.yaml: app y db. Docker Compose tomó estas definiciones y creó un contenedor para cada uno utilizando las imágenes de Docker especificadas (alexisfr/flask-app:latest para app y redis:alpine para db).

Docker Compose configuró la red para que los contenedores pudieran comunicarse entre sí. Creó una red personalizada (en mi caso, tp3_default) y conectó los contenedores a esta red. Esto es fundamental ya que mi aplicación Flask (app) depende de la base de datos Redis (db) para funcionar correctamente.

Para asegurarse de que los datos persistieran correctamente, Docker Compose gestionó la creación de volúmenes de datos. En este caso, se creó un volumen llamado tp3_redis_data para almacenar los datos de Redis de manera persistente, lo que significa que los datos no se perderán incluso si se detienen o eliminan los contenedores.

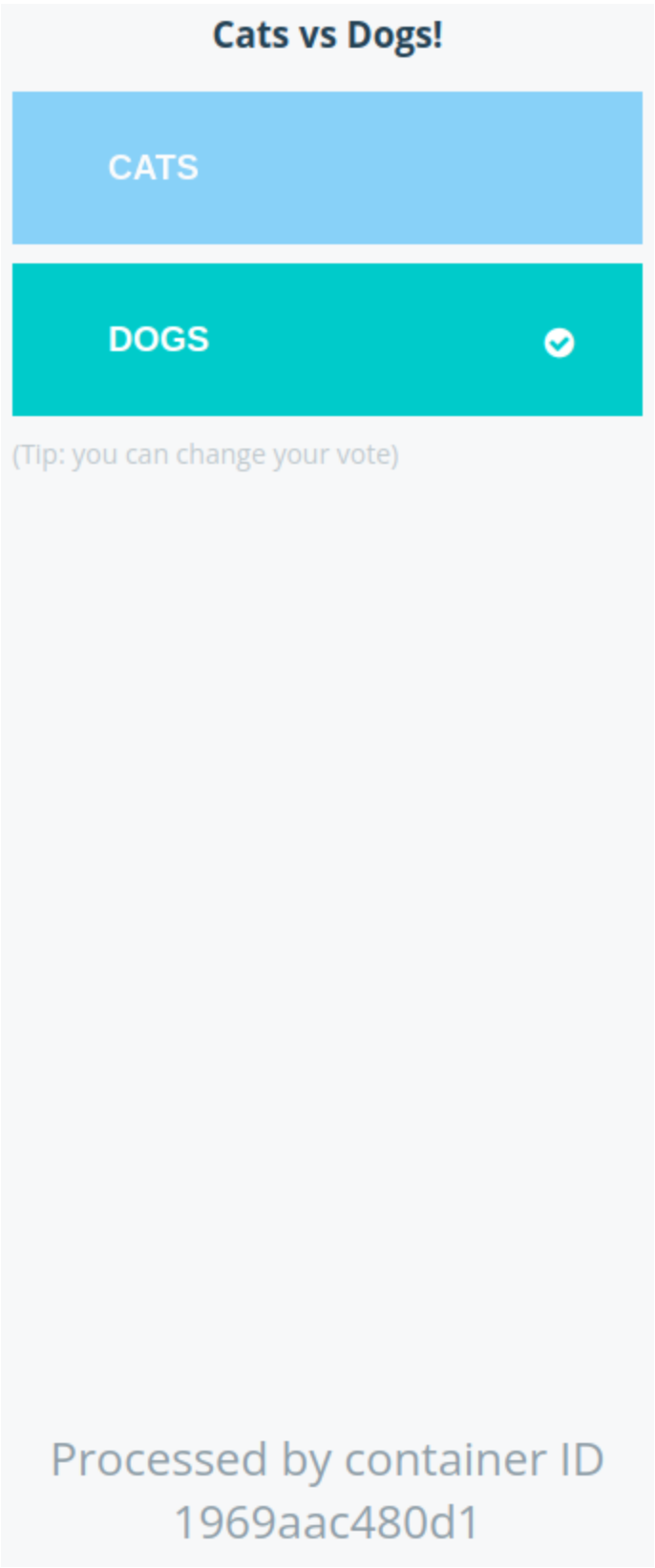
Docker Compose permitió la exposición de puertos. En el archivo docker-compose.yaml, especifiqué que el puerto 5000 del contenedor de la aplicación Flask (app) se debía mapear al puerto 5000 del host local. Esto significa que puedo acceder a mi aplicación Flask en mi navegador local a través de http://localhost:5000.

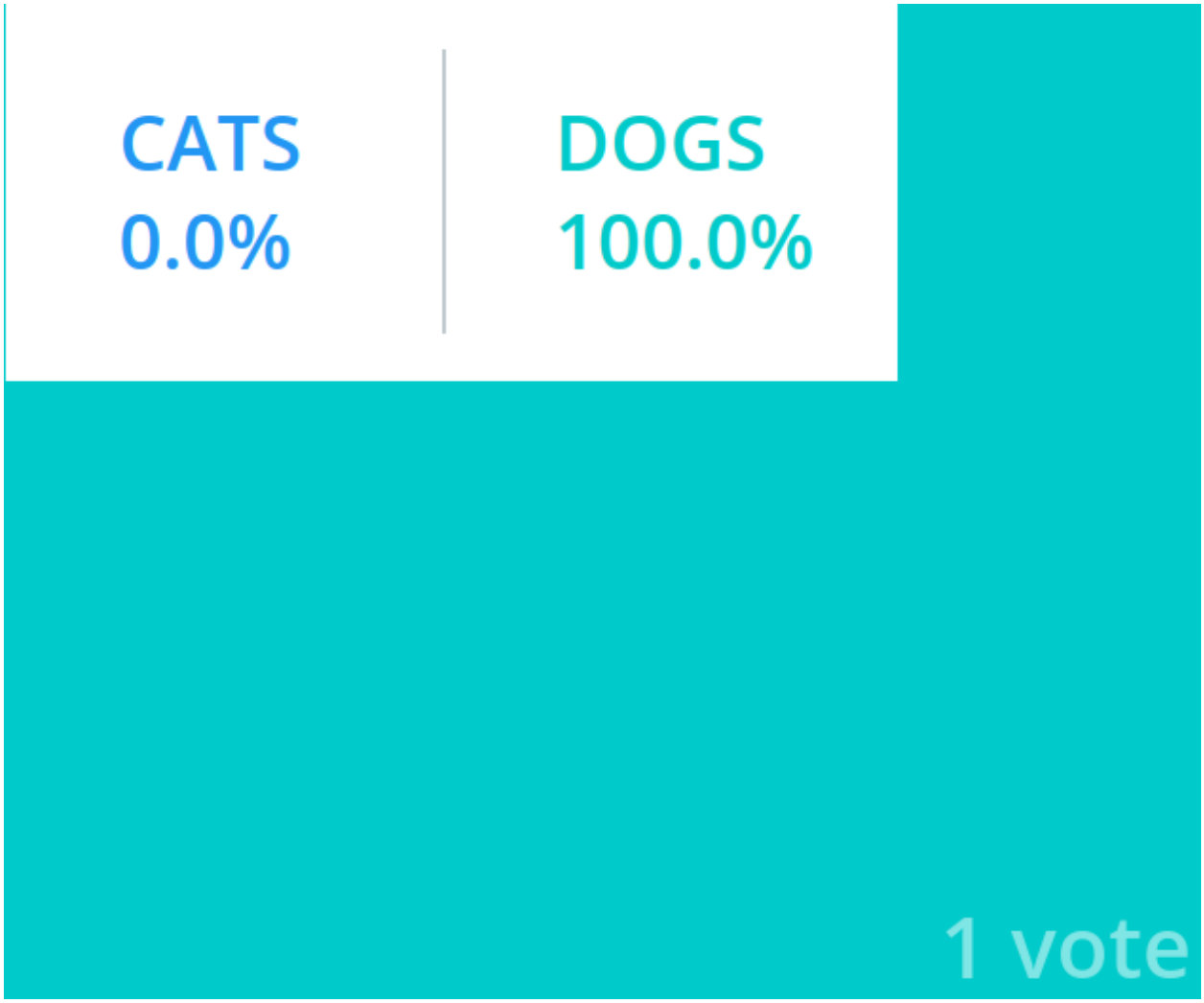


4- Aumentando la complejidad, análisis de otro sistema distribuido.

```
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$ git clone https://github.com/dockers
amples/example-voting-app
Cloning into 'example-voting-app'...
remote: Enumerating objects: 1099, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 1099 (delta 0), reused 1 (delta 0), pack-reused 1091
Receiving objects: 100% (1099/1099), 1.16 MiB | 280.00 KiB/s, done.
Resolving deltas: 100% (411/411), done.
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3$ cd example-voting-app/
tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3/example-voting-app$ docker-compose -f
 docker-compose.yml up -d
Creating network "example-voting-app_back-tier" with the default driver
Creating network "example-voting-app_front-tier" with the default driver
Creating volume "example-voting-app_db-data" with default driver
Building vote
[+] Building 136.6s (12/12) FINISHED                                                                docker:
desktop-linux
 => [internal] load build definition from Dockerfile
        0.0s
 => => transferring dockerfile: 740B
        0.0s
 => [internal] load .dockerignore
        0.0s
 => => transferring context: 2B
        0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim
       10.4s
 => [auth] library/python:pull token for registry-1.docker.io
        0.0s
 => [internal] load build context
        0.0s
 => => transferring context: 6.11kB
        0.0s
 => [1/6] FROM docker.io/library/python:3.9-slim@sha256:8a84bc20c838be617ba720f98a894d41c4fdaa8de27c2233b
9ed933  56.7s
 => => resolve docker.io/library/python:3.9-slim@sha256:8a84bc20c838be617ba720f98a894d41c4fdaa8de27c2233b
9ed9335  0.1s
 => => sha256:8a84bc20c838be617ba720f98a894d41c4fdaa8de27c2233b9ed9335fd061420 1.86kB / 1.86kB
        0.0s
 => => sha256:08a6a1666ddebe94becbec1986235cb8c321d2f7a7fd00f614befba5c1f23e67 1.37kB / 1.37kB
        0.0s
 => => sha256:cdecdc3a84699782600e00fde2c3e5067abdb3e4e1d560557fdc227b0c2b90b5 6.92kB / 6.92kB
        0.0s
 => => sha256:a803e7c4b030119420574a882a52b6431e160fceb7620f61b525d49bc2d58886 29.12MB / 29.12MB
       53.7s
 => => sha256:bf3336e84c8e00632cdea35b18fec9a5691711bdc8ac885e3ef54a3d5ff500ba 3.50MB / 3.50MB
        8.0s
 => => sha256:3614ca5053cfa002fa1e030c077def580221680f6278149d5d1b410af091431b 11.89MB / 11.89MB
       27.9s
 => => sha256:7f93433c11f3772ce686752ccd8e52fbb24c447eaeb22fcdbac00fdb5c3c6058 243B / 243B
        8.8s
 => => sha256:2fd2c896255c7556724e89e73c1ec921f1ba2b9f160c12538fbeac3b31503c78 3.13MB / 3.13MB
       17.1s
 => => extracting sha256:a803e7c4b030119420574a882a52b6431e160fceb7620f61b525d49bc2d58886
        1.4s
```

```
WARNING: Image for service result was built because it did not already exist. To rebuild this image you m
ust use `docker-compose build` or `docker-compose up --build`.
Creating example-voting-app_db_1    ... done

Creating example-voting-app_redis_1 ... done

Creating example-voting-app_vote_1   ... done

Creating example-voting-app_result_1 ... done

Creating example-voting-app_worker_1 ... done

tincho@IdeaPad-5-14ALC05:~/Documents/Ingenieria de Software III/TP3/example-voting-app$
```


Cats vs Dogs!
CATS
DOGS
(Tip: you can change your vote)
Processed by container ID
1969aac480d1



OBJ

Servicio Vote (Aplicación de Votación):
Este servicio utiliza una imagen definida en el directorio ./vote y ejecuta la aplicación Python app.py.
El servicio depende de Redis y se asegura de que Redis esté en un estado saludable antes de arrancar.
Se realizan comprobaciones de salud para asegurar que la aplicación esté funcionando correctamente.
Los archivos de la aplicación se montan desde el directorio ./vote en el contenedor.
El puerto 80 del contenedor se expone en el puerto 5000 del host.
Este servicio se conecta a las redes front-tier y back-tier.

Servicio Result (Aplicación de Resultados):
Este servicio utiliza una imagen definida en el directorio ./result y ejecuta la aplicación Node.js server.js con nodemon para desarrollo local.

Al igual que el servicio Vote, depende de la base de datos PostgreSQL y se inicia cuando la base de datos está saludable.
Los archivos de la aplicación se montan desde el directorio ./result en el contenedor.
Expone los puertos 80 y 5858 del contenedor en los puertos 5001 y 5858 del host.
Se conecta a las redes front-tier y back-tier.

Servicio Worker (Trabajador):
Este servicio utiliza un contexto definido en el directorio ./worker.
Dependiendo de Redis y PostgreSQL, se asegura de que ambos servicios estén saludables antes de iniciar.
Se conecta a la red back-tier.

Servicio Redis (Cola de Redis):
Utiliza la imagen de Redis en su versión alpina.
Monta archivos de comprobación de salud desde ./healthchecks en el contenedor y realiza comprobaciones de salud utilizando el script /healthchecks/redis.sh.
Este servicio se conecta a la red back-tier.

Servicio DB (Base de Datos PostgreSQL):
Utiliza la imagen de PostgreSQL en su versión alpina.
Se configuran las variables de entorno para el usuario y la contraseña de PostgreSQL.
Los datos de la base de datos se almacenan en un volumen llamado db-data.
Monta archivos de comprobación de salud desde ./healthchecks en el contenedor y realiza comprobaciones de salud utilizando el script /healthchecks/postgres.sh.
Se conecta a la red back-tier.

Servicio Seed (Inicialización de Datos):
Este servicio se ejecuta solo una vez para inicializar la base de datos con votos. Se activa cuando especificamos el perfil "seed" al ejecutar Docker Compose.
Depende del servicio Vote y se asegura de que el servicio de votación esté saludable antes de ejecutarse.
Este servicio se conecta a la red front-tier.

Volúmenes:
Se crea un volumen llamado db-data que se utiliza para persistir los datos de la base de datos PostgreSQL.

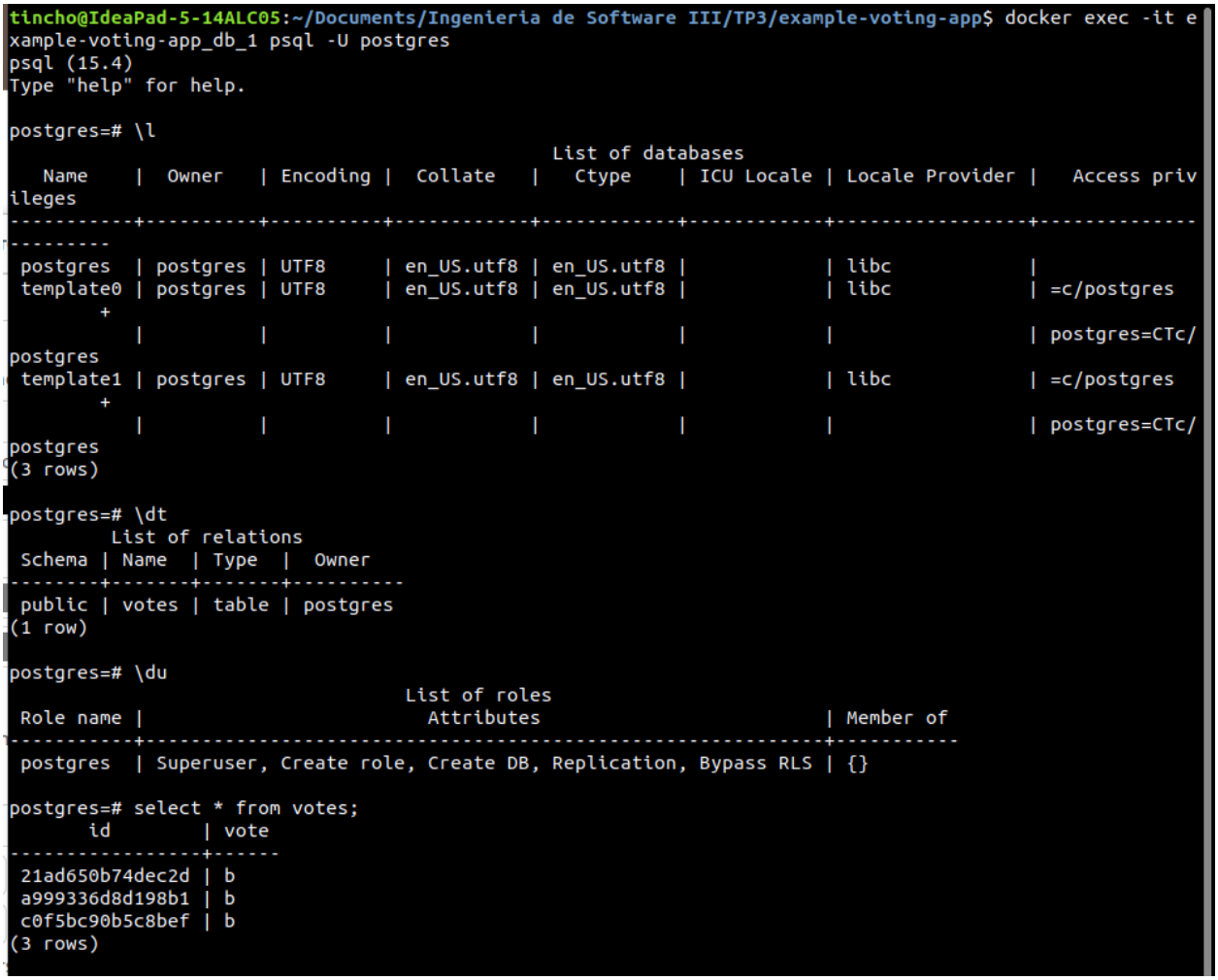
Redes:
Se definen dos redes, front-tier y back-tier, para asegurar que los servicios se comuniquen entre sí de manera adecuada.

5- Análisis detallado

Puertos expuestos y tablas de PostgreSQL

Open *docker-compose.yml ~/Documents/Ingenieria de Software III/TP3/example-voting-app Save

```yaml
      db:
        condition: service_healthy
    networks:
      - back-tier

  redis:
    image: redis:alpine
    ports:
    - "6380:6379"  # Exponiendo el puerto 6379 de Redis en el puerto 6380 del host
    volumes:
      - "./healthchecks:/healthchecks"
    healthcheck:
      test: /healthchecks/redis.sh
      interval: "5s"
    networks:
      - back-tier

  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"
    volumes:
      - "db-data:/var/lib/postgresql/data"
      - "./healthchecks:/healthchecks"
    healthcheck:
      test: /healthchecks/postgres.sh
      interval: "5s"
    networks:
      - back-tier

  # this service runs once to seed the database with votes
  # it won't run unless you specify the "seed" profile
  # docker compose --profile seed up -d
  seed:
    build: ./seed-data
    profiles: ["seed"]
```



app.py

Al comienzo del script, se definen algunas variables y se recogen valores de entorno. Por ejemplo, option_a y option_b representan las dos opciones de votación, que son "Cats" y "Dogs" de forma predeterminada. Además, se obtiene el nombre del host y se configura el registro de eventos.

Se define una función llamada get_redis que permite interactuar con la base de datos en memoria Redis. Esta función asegura que solo se cree una instancia de Redis por solicitud utilizando una característica llamada g de Flask. La conexión a Redis se establece utilizando el host "redis" y el puerto 6379.

La función hello maneja las solicitudes a la ruta principal del sitio web. Esta función es responsable tanto de las solicitudes GET como POST.
En primer lugar, se intenta obtener una cookie llamada "voter_id" que se utiliza para identificar a los votantes. Si no existe esta cookie, se genera una identificación única para el votante.

Cuando se realiza una solicitud POST, se procesa el voto. El voto seleccionado se recibe del formulario web y se almacena en la variable vote. Luego, se crea un objeto JSON que incluye la identificación única del votante y su elección de voto. Este objeto se almacena en Redis utilizando redis.rpush('votes', data) para agregarlo a una cola de votos.

Tras procesar el voto (si se emitió uno), se crea una respuesta HTML que muestra las dos opciones de votación, el nombre del host y el voto realizado. Esta respuesta se envía al navegador del usuario para su visualización.

Además de la respuesta HTML, se configura una cookie llamada "voter_id". Esto permite identificar al usuario en visitas posteriores al sitio web sin necesidad de emitir un nuevo voto.

program.cs
El código en program.cs es el núcleo del trabajador en el sistema de votación. Funciona estableciendo conexiones con dos componentes esenciales: PostgreSQL y Redis. Luego, entra en un bucle infinito que busca continuamente votos en la cola de Redis. Cuando se detecta un voto, se almacena en la base de datos PostgreSQL. Además, el código monitorea la conexión a Redis y PostgreSQL, y se reconecta automáticamente si alguna de ellas se pierde, asegurando un funcionamiento confiable del sistema incluso en condiciones adversas.

Para lograr esto, el programa utiliza manejo de excepciones para gestionar las conexiones y las transacciones con la base de datos, y emplea una consulta de "mantenimiento" en PostgreSQL para mantener la conexión activa.

server.js
El código en server.js es parte de la aplicación de votación y tiene dos funciones principales. Primero, establece un servidor web que escucha en un puerto y maneja las solicitudes de los usuarios. En segundo lugar, realiza un seguimiento de los votos en tiempo real desde la base de datos PostgreSQL y envía las actualizaciones a través de WebSocket a la página web para mostrar los resultados en tiempo real.

Utiliza Node.js y Express para configurar el servidor web y Socket.io para habilitar la comunicación en tiempo real. Se conecta a la base de datos PostgreSQL y recupera los votos, contando cuántos votos hay para cada opción ("a" y "b"). Luego, envía estos recuentos a la página web utilizando Socket.io, lo que permite que los resultados se actualicen automáticamente en la interfaz de usuario sin necesidad de recargar la página.