Trabajo Práctico N°3

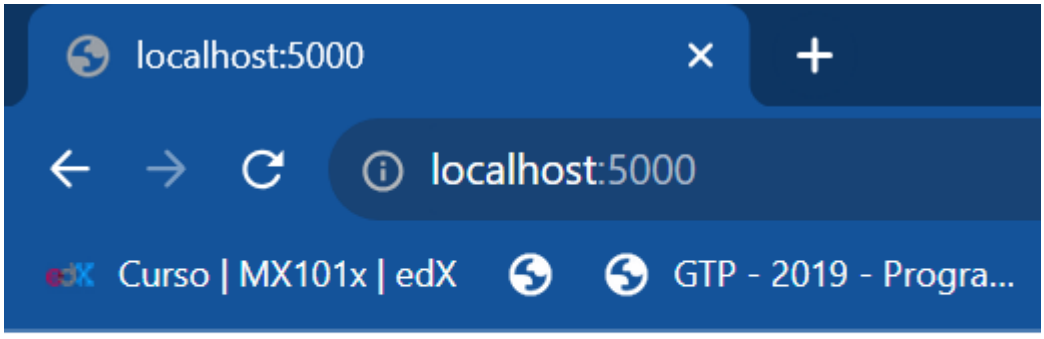
Arquitectura de Sistemas Distribuidos

Ejercicio 1

```
C:\Users\Usuario>docker network create -d bridge mybridge
799d4180979333496f840e546d83a0711d6f2763c54025d350bdcb5f7105d23d

C:\Users\Usuario> docker run -d --net mybridge --name db redis:alpine
Unable to find image 'redis:alpine' locally
alpine: Pulling from library/redis
7264a8db6415: Pull complete
a28817da73be: Pull complete
536ccaebaffb: Pull complete
f54d1871dea6: Pull complete
4d190b4b6472: Pull complete
33fcc95c965f: Pull complete
Digest: sha256:fd5de2340bc46cbc2241975ab027797c350dec6fd86349e3ac384e3a41be6fee
Status: Downloaded newer image for redis:alpine
7adfcdf154c45c5a25da66b86fd3bbbf8e7a69f7b8b80f43621da1eae58c5bcf

C:\Users\Usuario>  docker run -d --net mybridge -e REDIS_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
30042a88d8217b2dc5e2869b9b0773d4a929643c0506053051b1ee62b98a8041
```



Hello from Redis! I have been seen 2 times.

```
C:\Users\Usuario>docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED        STATUS        PORTS                                NAMES
30042a88d821   alexisfr/flask-app:latest  "python /app.py"         13 hours ago   Up 13 hours   0.0.0.0:5000->5000/tcp               web
7adfcdf154c4   redis:alpine               "docker-entrypoint.s…"   13 hours ago   Up 13 hours   6379/tcp                             db
96156e684459   mysql:latest               "docker-entrypoint.s…"   7 days ago     Up 7 days     33060/tcp, 0.0.0.0:3307->3306/tcp    mysql-server
2e8c10ff239c   postgres:9.4               "docker-entrypoint.s…"   7 days ago     Up 7 days     0.0.0.0:5432->5432/tcp               my-postgres

C:\Users\Usuario>docker network ls
NETWORK ID     NAME       DRIVER    SCOPE
1d4c8380db1c   bridge     bridge    local
839ed953d9f7   host       host      local
799d41809793   mybridge   bridge    local
927c671258cf   none       null      local
```

```
C:\Users\Usuario>docker network inspect mybridge
[
    {
        "Name": "mybridge",
        "Id": "799d4180979333496f840e546d83a0711d6f2763c54025d350bdcb5f7105d23d",
        "Created": "2023-08-21T23:27:10.867883Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "30042a88d8217b2dc5e2869b9b0773d4a929643c0506053051b1ee62b98a8041": {
                "Name": "web",
                "EndpointID": "ef01185e07098efb780ffb7746596dae6e11410d15b94905e826705ab4d4cd6f",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
            "7adfcdf154c45c5a25da66b86fd3bbbf8e7a69f7b8b80f43621da1eae58c5bcf": {
                "Name": "db",
                "EndpointID": "d48588f21e8a28e253adeb55d715d1e1ebc730ef140c88e032a0b22eea7b1a06",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
```

Ejercicio 2

Explicación del Sistema:

El sistema consiste en una aplicación web implementada en Flask que interactúa con una base de datos Redis. La aplicación recibe solicitudes HTTP a través de la ruta `'/'`, y cada vez que recibe una solicitud, incrementa un contador almacenado en la base de datos Redis. La aplicación luego muestra un mensaje en el navegador que indica cuántas veces se ha accedido a la página.

El código se conecta a la base de datos Redis utilizando la información de host y puerto proporcionada a través de variables de entorno. El parámetro `REDIS_HOST` indica la dirección del host de la base de datos Redis, y `REDIS_PORT` indica el puerto

en el que está escuchando. La variable `BIND_PORT` se utiliza para especificar el puerto en el que la aplicación Flask escuchará.

Parámetros -e en el segundo Docker run:

Los parámetros `-e` en el segundo `docker run` del ejercicio 1 se utilizan para pasar variables de entorno al contenedor. En este caso, los parámetros `-e REDIS_HOST=db -e REDIS_PORT=6379` se utilizan para proporcionar la dirección del host y el puerto de la base de datos Redis a la aplicación web dentro del contenedor. Esto permite que la aplicación web sepa cómo conectarse a la base de datos.

Efecto de ejecutar `docker rm -f web` y volver a correr el contenedor:

Al ejecutar `docker rm -f web`, se elimina el contenedor que aloja la aplicación web. Luego, cuando se vuelve a ejecutar `docker run` para crear un nuevo contenedor con la misma imagen, la aplicación web se inicia nuevamente y comienza a funcionar en el nuevo contenedor. Sin embargo, como el contador de visitas está almacenado en la base de datos Redis, si también se eliminó el contenedor de Redis, se perderá el contador de visitas acumulado.

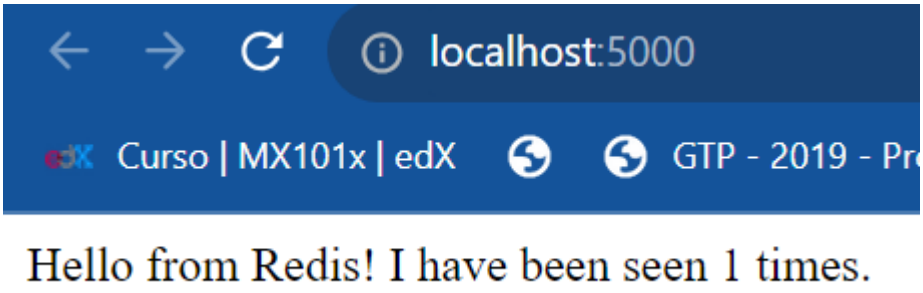
Efecto de borrar el contenedor de Redis y volver a correrlo:

Si se borra el contenedor de Redis con `docker rm -f db` y luego se vuelve a correr con `docker run`, se creará un nuevo contenedor Redis. Esto significa que se perderán todos los datos almacenados en el contenedor anterior, incluido el contador de visitas acumulado.

Para no perder la cuenta de las visitas:

Para no perder la cuenta de las visitas, se debe almacenar el contador en un lugar persistente fuera del contenedor, como en una base de datos externa o un servicio de almacenamiento en la nube. De esta manera, incluso si los contenedores se detienen o eliminan, los datos persistirán y se podrán seguir acumulando las visitas.

Ejercicio 3




```
PS C:\Users\Usuario\OneDrive\Escritorio\2023\Ing SW 3\tp3> docker ps
1c9f6a350229   alexisfr/flask-app:latest   "python /app.py"         About a minute ago   Up About a minute   0.0.0.0:5000->5000/tcp   tp3-app-1
b0f54e50b239   redis:alpine                "docker-entrypoint.s…"   About a minute ago   Up About a minute   6379/tcp                 tp3-db-1
PS C:\Users\Usuario\OneDrive\Escritorio\2023\Ing SW 3\tp3> docker network ls
NETWORK ID     NAME          DRIVER    SCOPE
1d4c8380db1c   bridge        bridge    local
839ed953d9f7   host          host      local
6314ff6d583d   mybridge      bridge    local
927c671258cf   none          null      local
619b7603e4fe   tp3_default   bridge    local
PS C:\Users\Usuario\OneDrive\Escritorio\2023\Ing SW 3\tp3> docker volume ls
DRIVER    VOLUME NAME
local     tp3_redis_data
```

Docker Compose automatiza la creación, configuración e interconexión de múltiples contenedores según las definiciones proporcionadas en el archivo YAML. Facilita la administración de aplicaciones multi-contenedor y garantiza que los servicios dependientes se ejecuten en el orden correcto.

Definición de Servicios y Configuraciones:

En el archivo `docker-compose.yaml`, se definen dos servicios (`app` y `db`) junto con sus imágenes, dependencias, variables de entorno, puertos y otras configuraciones.

Creación de Contenedores:

Al ejecutar `docker-compose up -d`, Docker Compose interpretará el archivo YAML y generará los contenedores conforme a las definiciones. Se instanciará un contenedor para la aplicación web y otro para la base de datos Redis. El parámetro `-d` indica que los contenedores funcionarán en segundo plano.

Interconexión Automática:

La sección `depends_on` en la definición del servicio `app` establece la dependencia del servicio `app` con respecto al servicio `db`. Docker Compose orquesta el inicio de los contenedores en el orden preciso, asegurándose de que el contenedor `db` esté en funcionamiento antes de iniciar `app`.

Variables de Entorno:

Las variables de entorno como `REDIS_HOST` y `REDIS_PORT` son definidas para el servicio `app` en el archivo YAML. Docker Compose transmitirá automáticamente estas variables a los contenedores, posibilitando que la aplicación web se comunique con la base de datos.

Mapeo de Puertos:

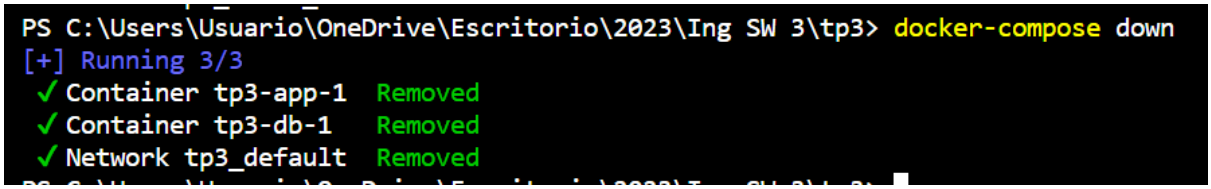
El mapeo de puertos también es especificado en el archivo YAML. Docker Compose asigna el puerto 5000 del contenedor `app` al puerto 5000 del host, permitiendo el acceso a la aplicación web mediante http://localhost:5000/.

Volúmenes Definidos:

La definición del servicio `db` incluye un volumen denominado `redis_data`. Esto asegura que los datos de la base de datos Redis persistan inclusive después de detener o remover el contenedor. Docker Compose se encarga de establecer y administrar este volumen.

Simplificación de Comandos:

Docker Compose unifica todos estos elementos en un único comando: `docker-compose up -d`. Este enfoque simplifica considerablemente el procedimiento de establecimiento y gestión de contenedores interconectados y sus configuraciones.



Ejercicio 4

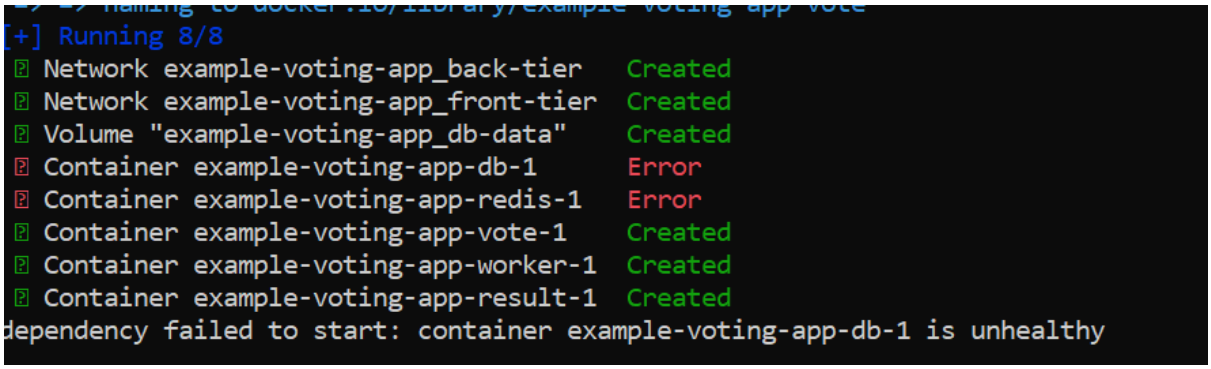
1. **Servicio `vote`:**
    - Se construye a partir del directorio `./vote`.
    - El comando para ejecutar el servicio es `python app.py`.
    - Depende del servicio `redis`, y espera a que esté en un estado saludable antes de iniciarse.
    - Se realiza un chequeo de salud con el comando `curl -f <http://localhost>` cada 15 segundos. Si el comando falla, se intentará durante un período de tiempo determinado.
    - Los archivos en `./vote` se montan en la ubicación `/app` dentro del contenedor.
    - Se mapea el puerto `5000` del host al puerto `80` del contenedor.
    - Pertenece a las redes `front-tier` y `back-tier`.
2. **Servicio `result`:**

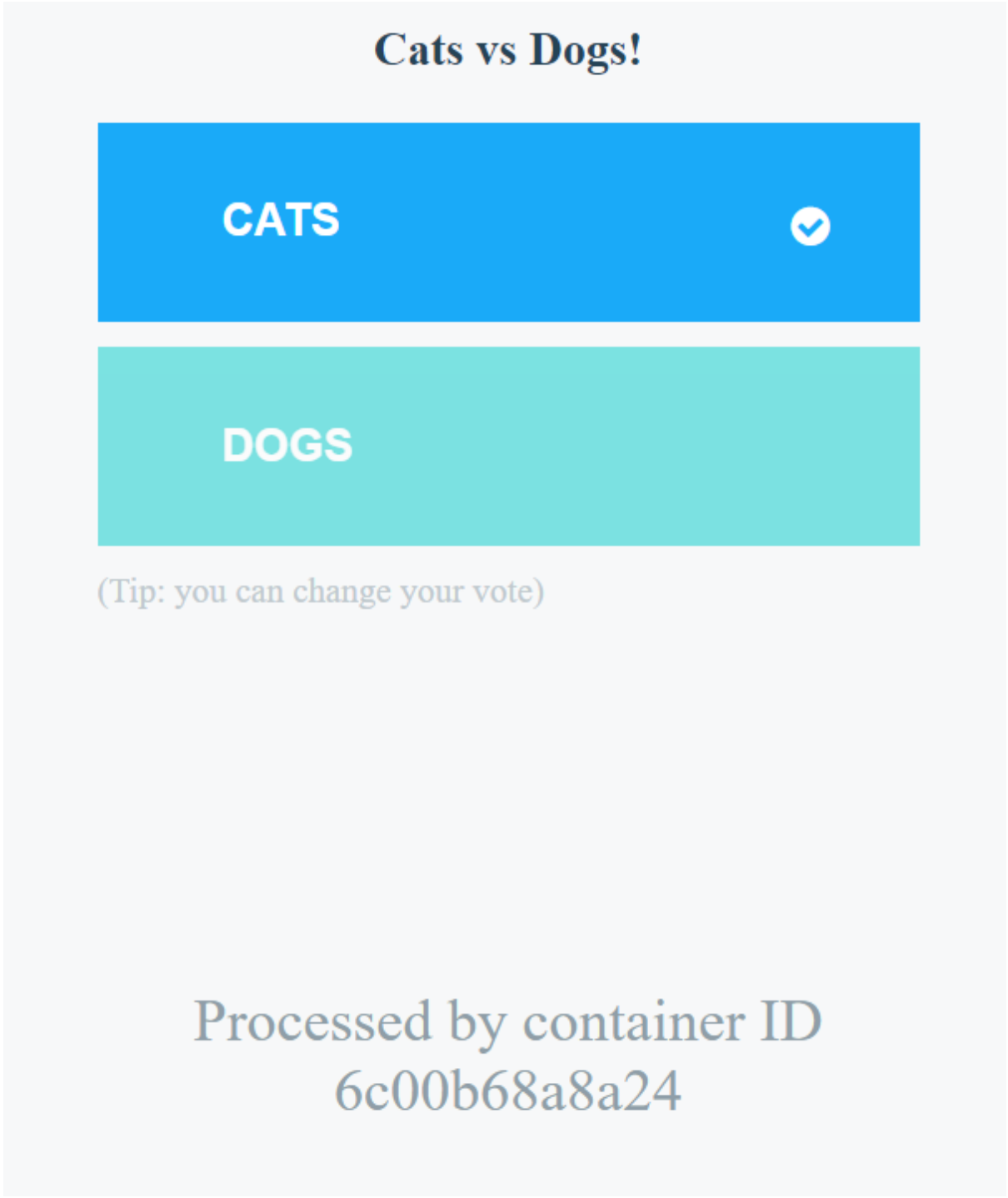
- Se construye a partir del directorio `./result`.
- El comando de entrada es `nodemon server.js`.
- Depende del servicio `db`, y espera a que esté en un estado saludable antes de iniciarse.
- Se montan los archivos en `./result` en la ubicación `/app` dentro del contenedor.
- Se mapean los puertos `5001` y `5858` del host a los puertos `80` y `5858` del contenedor respectivamente.
- Pertenece a las redes `front-tier` y `back-tier`.

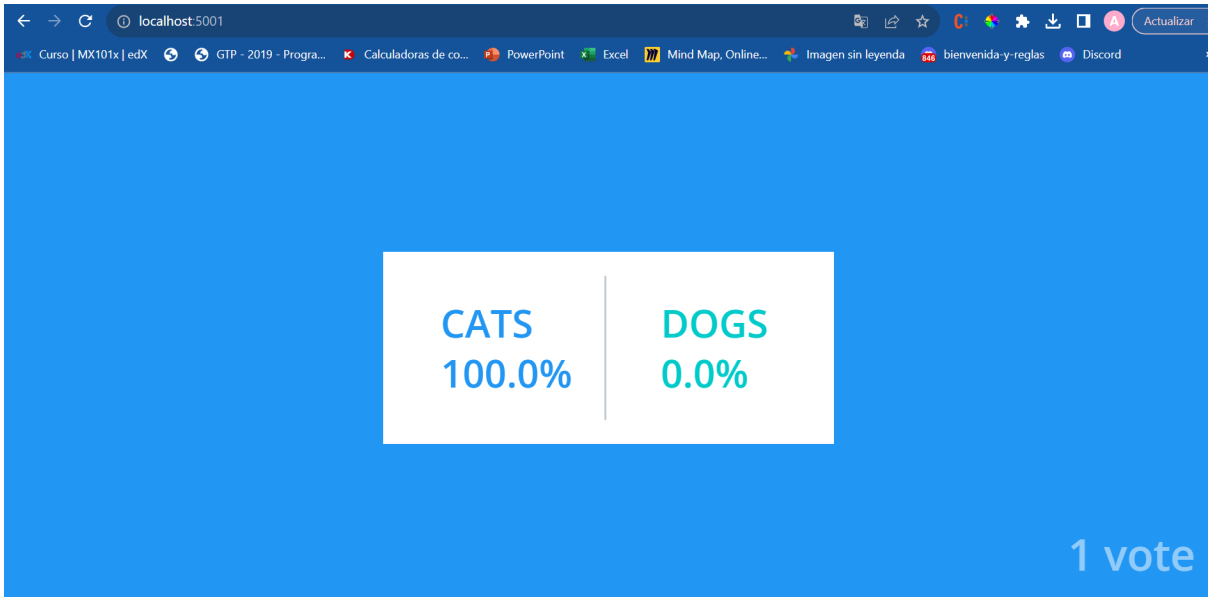
3. **Servicio `worker`:**
   - Se construye a partir del directorio `./worker`.
   - Depende de los servicios `redis` y `db`, y espera a que ambos estén en un estado saludable antes de iniciarse.
   - Pertenece a la red `back-tier`.
4. **Servicio `redis`:**
   - Utiliza la imagen `redis:alpine`.
   - Se montan los archivos en `./healthchecks` en el contenedor en la ubicación `/healthchecks`.
   - Se realiza un chequeo de salud con el script `redis.sh` cada 5 segundos.
   - Pertenece a la red `back-tier`.
5. **Servicio `db`:**
   - Utiliza la imagen `postgres:15-alpine`.
   - Define las variables de entorno `POSTGRES_USER` y `POSTGRES_PASSWORD`.
   - Se montan los volúmenes `db-data` y `./healthchecks` en el contenedor.
   - Se realiza un chequeo de salud con el script `postgres.sh` cada 5 segundos.
   - Pertenece a la red `back-tier`.
6. **Servicio `seed`:**
   - Se construye a partir del directorio `./seed-data`.
   - Se ejecuta solo si se especifica el perfil `seed`.

- Depende del servicio `vote`, y espera a que esté en un estado saludable antes de iniciarse.
- Pertenece a la red `front-tier`.
- No se reinicia automáticamente.

7. **Volumen `db-data`:**
   - Utilizado por el servicio `db` para almacenar los datos de la base de datos PostgreSQL.
8. **Redes `front-tier` y `back-tier`:**
   - Son redes a las que se conectan los servicios para facilitar la comunicación entre ellos.



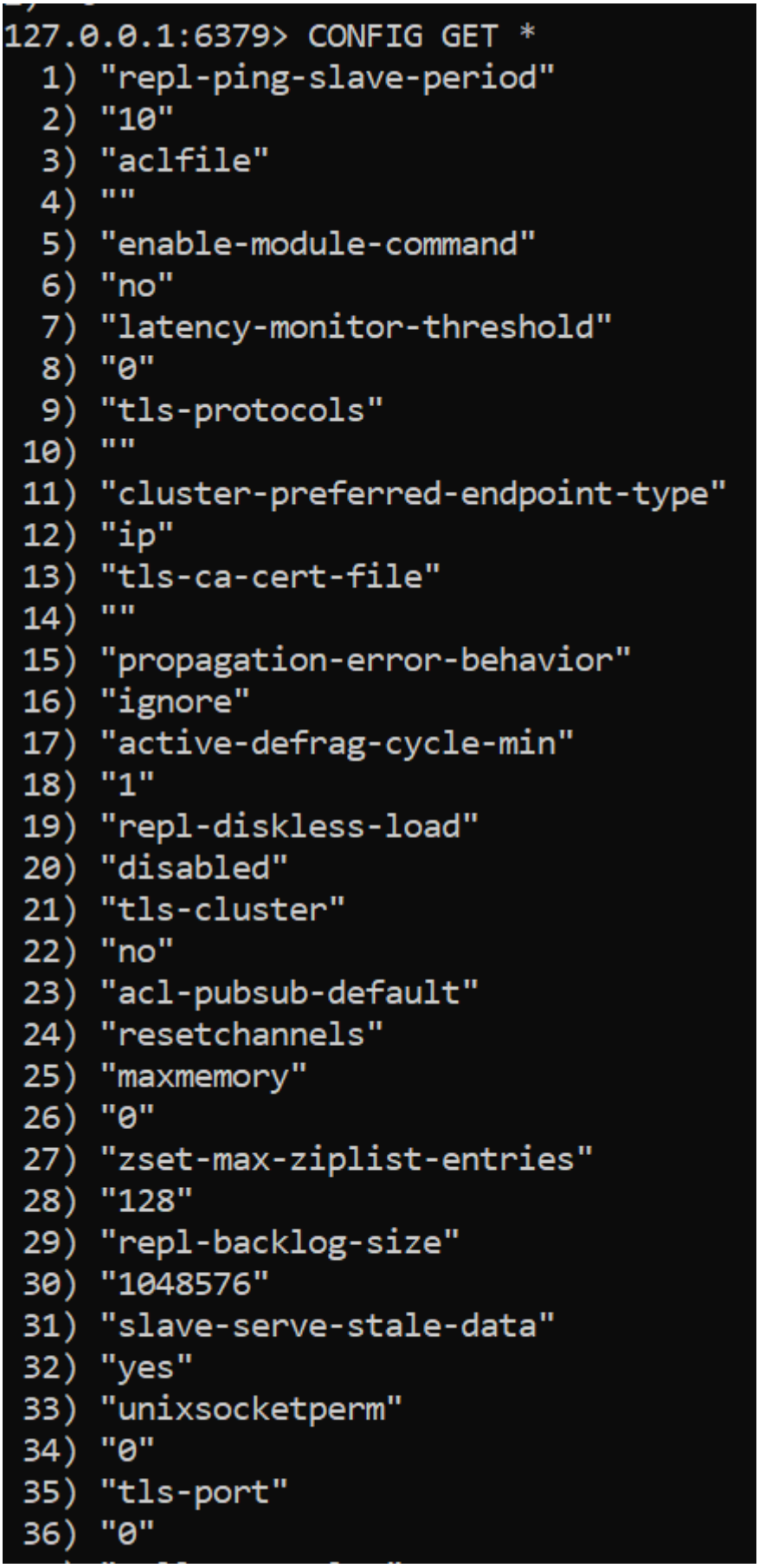

Cats vs Dogs!
CATS
DOGS
(Tip: you can change your vote)
Processed by container ID
6c00b68a8a24



Redis:

```
Usuario@DESKTOP-8G1HL0R MINGW64 ~/OneDrive/Escritorio/2023/Ing SW 3/example-voting-app (main)
$ docker exec -it 4d6e56e7f3a9 redis-cli
127.0.0.1:6379> scan 0
1) "0"
2) (empty array)
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379> SET clave valorclaveprueba
OK
127.0.0.1:6379> GET clave
"valorclaveprueba"
127.0.0.1:6379>
```

```yaml
redis:
  image: redis:alpine
  ports:
    - "6379:6379"  # Mapea el puerto 6379 del host al puerto 6379 del contenedor
  volumes:
    - "./healthchecks:/healthchecks"
  healthcheck:
    test: /healthchecks/redis.sh
    interval: "5s"
  networks:
    - back-tier
```


127.0.0.1:6379> CONFIG GET *
1) "repl-ping-slave-period"
2) "10"
3) "aclfile"
4) ""
5) "enable-module-command"
6) "no"
7) "latency-monitor-threshold"
8) "0"
9) "tls-protocols"
10) ""
11) "cluster-preferred-endpoint-type"
12) "ip"
13) "tls-ca-cert-file"
14) ""
15) "propagation-error-behavior"
16) "ignore"
17) "active-defrag-cycle-min"
18) "1"
19) "repl-diskless-load"
20) "disabled"
21) "tls-cluster"
22) "no"
23) "acl-pubsub-default"
24) "resetchannels"
25) "maxmemory"
26) "0"
27) "zset-max-ziplist-entries"
28) "128"
29) "repl-backlog-size"
30) "1048576"
31) "slave-serve-stale-data"
32) "yes"
33) "unixsocketperm"
34) "0"
35) "tls-port"
36) "0"

PostgreSQL:

No funciona la conexión desde DBeaver

```
Usuario@DESKTOP-8G1HL0R MINGW64 ~/OneDrive/Escritorio/2023/Ing SW 3/example-voting-app (main)
$ docker exec -it d4b5e049f8fc psql -h localhost -U postgres
psql (15.4)
Type "help" for help.

postgres=# \dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | votes | table | postgres
(1 row)

postgres=# SELECT * FROM votes;
       id        | vote
-----------------+------
 ef52ccc06bf5455 | a
(1 row)
```

Revisar el código de la aplicación Python `example-voting-app\vote\app.py` ***para ver como envía votos a Redis.***

Aplicación web Flask.

1. Importa las librerías necesarias para crear la aplicación Flask, trabajar con Redis, gestionar el registro y obtener información sobre el sistema.
2. Obtiene las opciones de voto A y B desde variables de entorno (si no se especifican, se usan valores predeterminados). Obtiene el nombre del host del sistema.
3. get_redis(): Define una función para obtener una instancia de Redis. Utiliza un atributo `g` de Flask para mantener una instancia de Redis por solicitud.
4. Si el método de solicitud es POST, significa que se envió un voto.
    a. Obtiene la instancia de Redis utilizando la función get_redis.
    b. Registra el voto recibido.
    c. Convierte los datos del voto en formato **JSON** y **los agrega a la lista votes en Redis utilizando rpush.**
    d. Genera un identificador único para el votante si no se ha proporcionado ya (usando cookies).
5. Renderiza la plantilla `index.html` con las opciones, el nombre del host y el voto actual.

Revisar el código del worker `example-voting-app\worker\program.cs` para entender como procesa los datos.

El worker es responsable de procesar los votos almacenados en Redis y actualizar una base de datos PostgreSQL con los resultados de los votos.

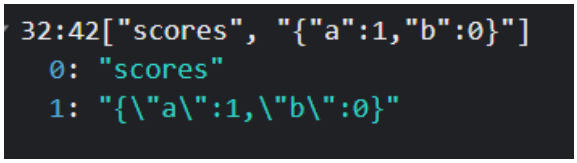
1. Se realiza un bucle infinito para continuar procesando los votos.
2. La función `OpenDbConnection` maneja la apertura de la conexión a la base de datos PostgreSQL. Intenta conectar a la base de datos en un bucle y espera si hay excepciones de conexión.
3. Después de conectarse a la base de datos, se ejecuta una consulta para crear la tabla `votes` si no existe.
4. La función `OpenRedisConnection` maneja la conexión a Redis. Intenta conectarse a Redis en un bucle y espera si hay excepciones de conexión.
5. Se utiliza la función `GetIp` para obtener la dirección IP de Redis.
6. **Procesamiento de Votos:**
    - En el bucle infinito, se obtiene un voto de Redis utilizando `ListLeftPopAsync`.
    - Si se encuentra un voto, se procesa. Se muestra información sobre el voto y se actualiza la base de datos con ese voto.
    - Si no hay votos en la cola, se ejecuta un comando de mantenimiento (`keepAliveCommand`).
    - La función `UpdateVote` actualiza la base de datos con el voto proporcionado.
    - Se intenta insertar un nuevo voto. Si ya existe un voto para el mismo votante, se actualiza.

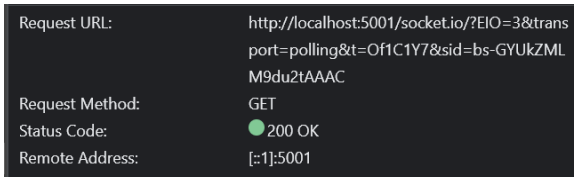
💡 Una vez que un voto ha sido procesado por el worker, se elimina automáticamente de la cola de Redis. En el código se utiliza el comando `ListLeftPopAsync` para obtener y eliminar el voto procesado de la lista `votes` en Redis. Esto significa que después de que el voto haya sido procesado, ya no estará en la cola y no se podrá acceder a él directamente desde Redis. El comando `ListLeftPopAsync` se utiliza para realizar una operación de "pop" desde la izquierda de la lista. Esto significa que toma el primer elemento de la lista, lo elimina de la lista y lo devuelve. En este caso, el elemento que se está "pop" es el JSON que contiene la información del voto.

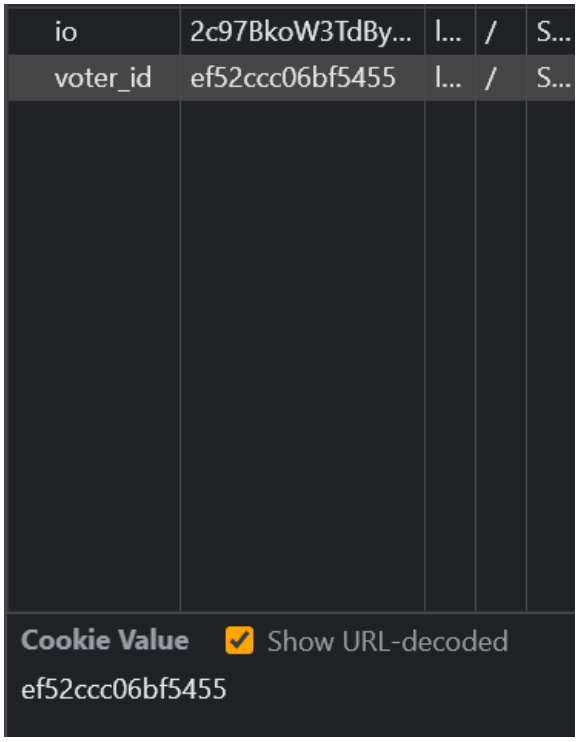
Revisar el código de la aplicacion que muestra los resultados `example-voting-app\result\server.js` para entender como muestra los valores.

Servidor de la aplicación de resultados que muestra los resultados de los votos en tiempo real utilizando Socket.IO.

1. Configura `socket.io` para usar el transporte de "polling" en lugar de WebSockets.
2. Crea una instancia del grupo `pg.Pool` para manejar las conexiones a la base de datos PostgreSQL. Utiliza `async.retry` para intentar conectarse a la base de datos varias veces con intervalos si no se puede conectar inmediatamente.
3. `getVotes`:Esta función consulta la base de datos para obtener los votos y sus recuentos. Luego, emite los resultados a través de `socket.io` al canal "scores" en formato JSON.
4. `collectVotesFromResult`**:** Esta función procesa los resultados de la consulta a la base de datos y crea un objeto con los recuentos de votos.







Documento de Arquitectura: Sistema de Votación en Tiempo Real

Introducción

Este documento describe la arquitectura de un sistema de votación en tiempo real basado en contenedores Docker y utilizando tecnologías como Flask, Redis, PostgreSQL, y Socket.IO. El sistema permite a los usuarios votar entre dos opciones y muestra los resultados en tiempo real a través de una interfaz web.

Componentes del Sistema

El sistema consta de varios componentes que trabajan en conjunto para permitir la votación y la visualización de resultados en tiempo real.

1. Aplicación de Votación (Vote)

- Descripción: Esta es la interfaz web que permite a los usuarios votar entre dos opciones (opción A y opción B).
- Tecnologías: Flask (Python), Redis.

- Interacción:
  - Los usuarios acceden a la interfaz web y seleccionan su voto.
  - La aplicación almacena los votos en Redis en forma de objetos JSON.
  - Los votos se encolan en la lista "votes" en Redis.

2. Cola de Votos (Redis)

- Descripción: Redis se utiliza como una cola para almacenar los votos en forma de objetos JSON.
- Tecnología: Redis.
- Interacción:
  - La aplicación de votación encola los votos en Redis en formato JSON.
  - El worker retira los votos de la cola para su procesamiento.

3. Worker

- Descripción: El worker procesa los votos almacenados en Redis y los actualiza en la base de datos.
- Tecnologías: C# (worker), Npgsql (PostgreSQL), StackExchange.Redis (Redis).
- Interacción:
  - El worker consulta la cola de Redis para obtener los votos.
  - Procesa los votos y los actualiza en la base de datos PostgreSQL.
  - Elimina los votos procesados de la cola de Redis.

4. Base de Datos (PostgreSQL)

- Descripción: La base de datos almacena los votos procesados y sus recuentos.
- Tecnología: PostgreSQL.
- Interacción:
  - El worker actualiza la base de datos con los votos procesados.
  - La aplicación de resultados consulta la base de datos para mostrar los recuentos de votos.

5. Aplicación de Resultados (Result)

- Descripción: La aplicación web muestra los resultados de los votos en tiempo real.
- Tecnologías: Express (Node.js), Socket.IO, pg (PostgreSQL).
- Interacción:
  - La aplicación de resultados consulta la base de datos PostgreSQL para obtener los recuentos de votos.
  - Utiliza Socket.IO para transmitir los resultados a los clientes en tiempo real.

Interacción entre Componentes

1. Los usuarios votan a través de la interfaz web de la aplicación de votación.
2. Los votos se almacenan en Redis como objetos JSON en la lista "votes".
3. El worker recoge los votos de Redis, los procesa y los actualiza en la base de datos PostgreSQL.
4. La aplicación de resultados consulta la base de datos para obtener los recuentos de votos.
5. Los resultados se transmiten a través de Socket.IO a los clientes conectados en la aplicación de resultados.

Docker Compose y Funcionamiento de Docker

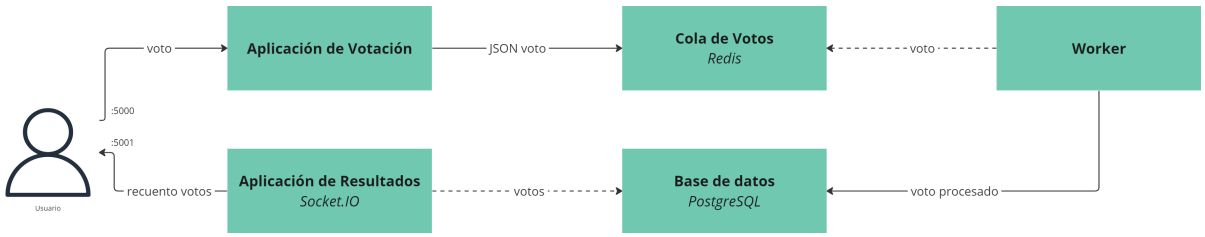
- Docker Compose: Se utiliza Docker Compose para definir y orquestar los servicios que componen la aplicación. El archivo `docker-compose.yml` especifica cómo se deben construir los servicios, qué contenedores se deben ejecutar y cómo se conectan entre sí.
- Funcionamiento de Docker:
  - Cada componente (votación, resultados, worker, Redis, PostgreSQL) se define como un servicio en el archivo `docker-compose.yml`.
  - Los servicios se construyen y se ejecutan en contenedores aislados.
  - Los contenedores están conectados a redes definidas en Docker Compose para facilitar la comunicación entre ellos.
  - Los volúmenes son utilizados para persistir los datos de Redis y PostgreSQL.

Beneficios y Escalabilidad

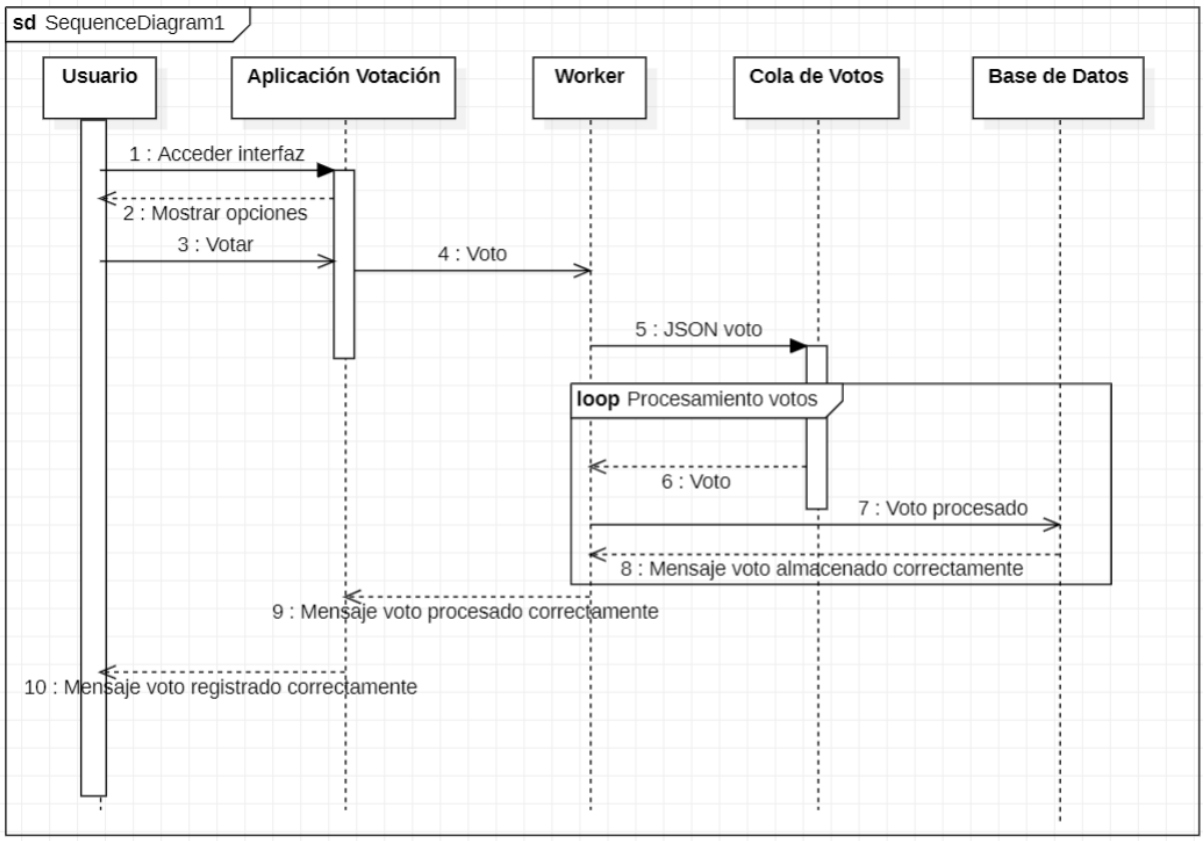
- La arquitectura basada en contenedores permite un despliegue y escalabilidad eficiente.
- Los votos se procesan en segundo plano, evitando bloqueos en la interfaz de votación.
- Los resultados se actualizan en tiempo real, brindando una experiencia interactiva a los usuarios.
- Redis actúa como una cola, lo que permite lidiar con cargas de trabajo fluctuantes.
- Docker simplifica la gestión de la infraestructura y la implementación.
- Los contenedores aíslan los componentes, lo que mejora la seguridad y la portabilidad.
- Docker Compose facilita la configuración y el despliegue de múltiples servicios.
- La arquitectura es escalable, permitiendo agregar más instancias según sea necesario.

Conclusiones

La arquitectura descrita proporciona un sistema de votación en tiempo real eficiente y escalable. Los componentes trabajan en conjunto para permitir la votación, el procesamiento de votos y la visualización de resultados actualizados en tiempo real. La combinación de Flask, C#, Express, Redis y PostgreSQL proporciona una solución robusta y flexible para el caso de uso de votación en tiempo real.



Registrar un voto:



Ver resultados: