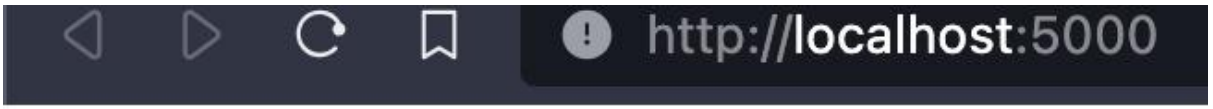
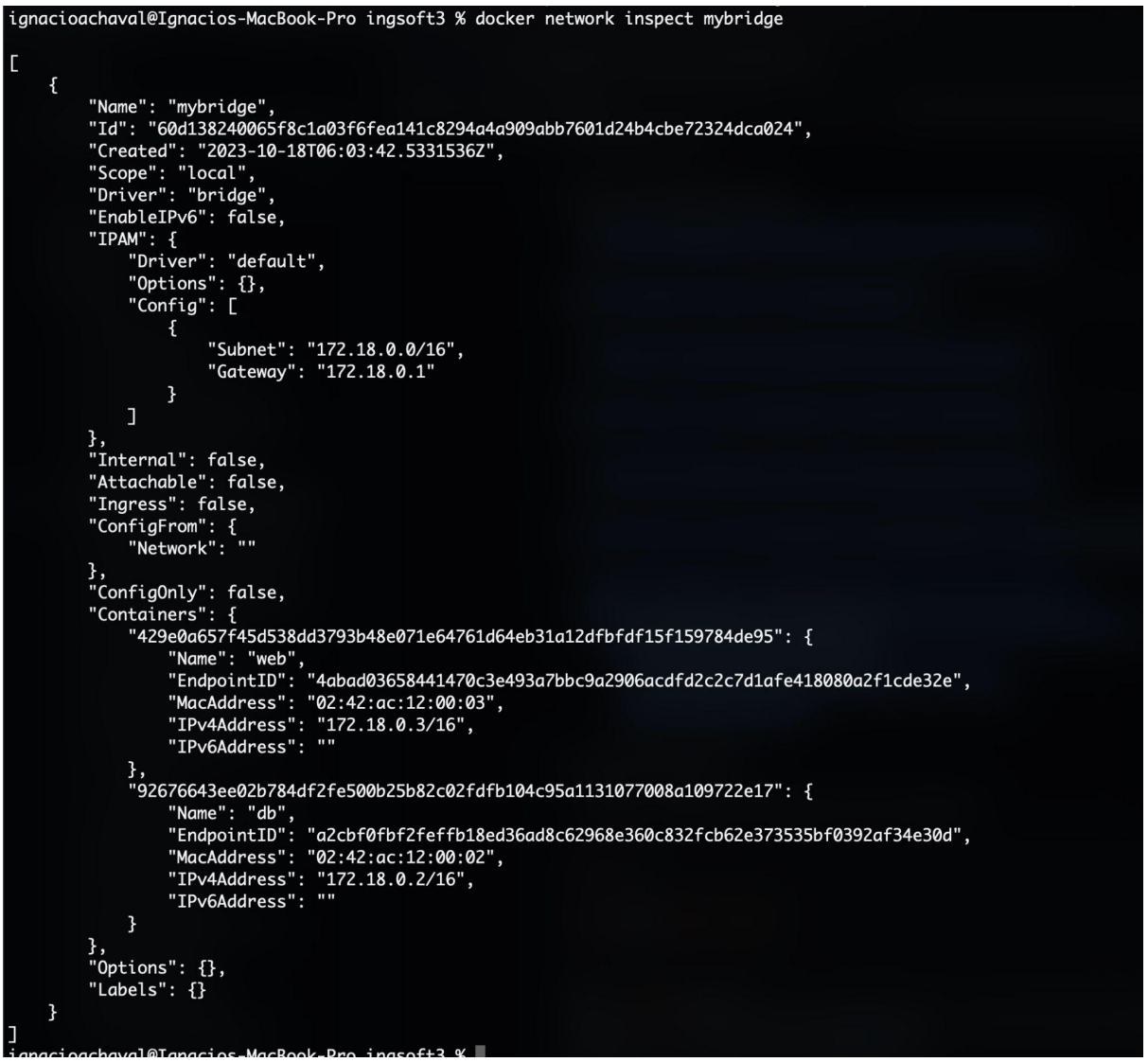
1)

```
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker network create -d bridge mybridge
60d138240065f8c1a03f6fea141c8294a4a909abb7601d24b4cbe72324dca024
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 %  docker run -d --net mybridge --name db redis:alpine
Unable to find image 'redis:alpine' locally
alpine: Pulling from library/redis
96526aa774ef: Pull complete
126607e66bec: Pull complete
20426db6bdc4: Pull complete
1c389a66bd59: Pull complete
0c2dcb3c0f65: Pull complete
4f4fb700ef54: Pull complete
06965e193854: Pull complete
Digest: sha256:a4c910678a1b358a88446b2fe1c5bb0417a5c9cfac4eba959f1e42d22f2e0af1
Status: Downloaded newer image for redis:alpine
92676643ee02b784df2fe500b25b82c02fdfb104c95a1131077008a109722e17
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 %   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
429e0a657f45d538dd3793b48e071e64761d64eb31a12dfbfdf15f159784de95
```



Hello from Redis! I have been seen 1 times.

```
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED          STATUS          PORTS                                       NAMES
429e0a657f45   alexisfr/flask-app:latest   "python /app.py"         3 minutes ago    Up 3 minutes    0.0.0.0:5000->5000/tcp                      web
92676643ee02   redis:alpine               "docker-entrypoint.s…"   4 minutes ago    Up 4 minutes    6379/tcp                                    db
3ff5065fcd5b   postgres:9.4               "docker-entrypoint.s…"   24 minutes ago   Up 24 minutes   0.0.0.0:5432->5432/tcp                      my-postgres
f30cb9e70d74   350c60fdbbaa               "dotnet SimpleWebAPI…"   2 hours ago      Up 2 hours      443/tcp, 5254/tcp, 0.0.0.0:8080->80/tcp     awesome_ramanujan
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker network inspect mybridge

[
```



2)

El códigoes una aplicación web escrita en Python utilizando el framework Flask. Esta aplicación web se comunica con una base de datos Redis para realizar un seguimiento del número de visitas y mostrar un mensaje en función de cuántas veces se ha visitado el sitio. Aquí está una explicación de cómo funciona este sistema:

Importación de módulos:
El código comienza importando los módulos necesarios, incluyendo os, Flask y Redis. Estos módulos se utilizan para trabajar con variables de entorno, crear la aplicación web y comunicarse con la base de datos Redis.

Creación de la aplicación Flask y conexión a Redis:
Se crea una instancia de la aplicación Flask con app = Flask(__name__). Luego, se establece una conexión con la base de datos Redis utilizando las variables de entorno REDIS_HOST y REDIS_PORT para especificar la dirección del host y el puerto de Redis.

Definición de una Ruta y Función de Visualización:
La aplicación define una sola ruta en la raíz ("/") y una función asociada llamada hello(). Cuando un usuario accede a la URL raíz, esta función se ejecuta.

Función hello():
La función hello() realiza las siguientes acciones:

Incrementa el contador de visitas en Redis con redis.incr('hits'). Esto aumenta el valor asociado a la clave 'hits' en Redis en 1.
Recupera el valor actual del contador de visitas desde Redis con total_hits = redis.get('hits').decode(). La función decode() se utiliza para convertir el valor almacenado en Redis a una cadena de texto.
Retorna un mensaje al navegador del usuario que incluye la cantidad total de visitas hasta ese momento.
Configuración del Servidor Flask:
La última sección del código verifica si el archivo se ejecuta directamente (es decir, no se importa como un módulo). Si se ejecuta directamente, la aplicación Flask se inicia con la función app.run(). La aplicación se ejecuta en "0.0.0.0", lo que significa que está disponible en todas las interfaces de red, y en el puerto especificado en la variable bind_port.

En resumen, esta aplicación web utiliza Flask para crear un servidor web simple que muestra un mensaje personalizado basado en cuántas veces se ha visitado la página. La información sobre el número de visitas se almacena en una base de datos Redis. Cada vez que se visita la página, se incrementa el contador de visitas en Redis, y la página muestra el número total de visitas.

Los parámetros -e en el comando docker run se utilizan para establecer variables de entorno en el contenedor. En el contexto de la aplicación web que se está ejecutando, estas variables de entorno se utilizan para configurar la conexión a la base de datos Redis.

Si se ejecuta docker rm -f web para eliminar el contenedor llamado "web" y luego se vuelve a ejecutar docker run con los mismos parámetros para crear un nuevo contenedor con el mismo nombre "web", ocurrirán los siguientes efectos:

Eliminación del Contenedor Existente, Creación de un Nuevo Contenedor, Cambios en el Contador de Visitas:

Dado que el contador de visitas se almacena en la base de datos Redis, la eliminación del contenedor "web" no afectará el contador de visitas en Redis. Los datos en Redis seguirán intactos.

Cuando eliminas el contenedor de Redis con docker rm -f db, la base de datos Redis se detiene y todos los datos que estaban almacenados en esa instancia de Redis se pierden.

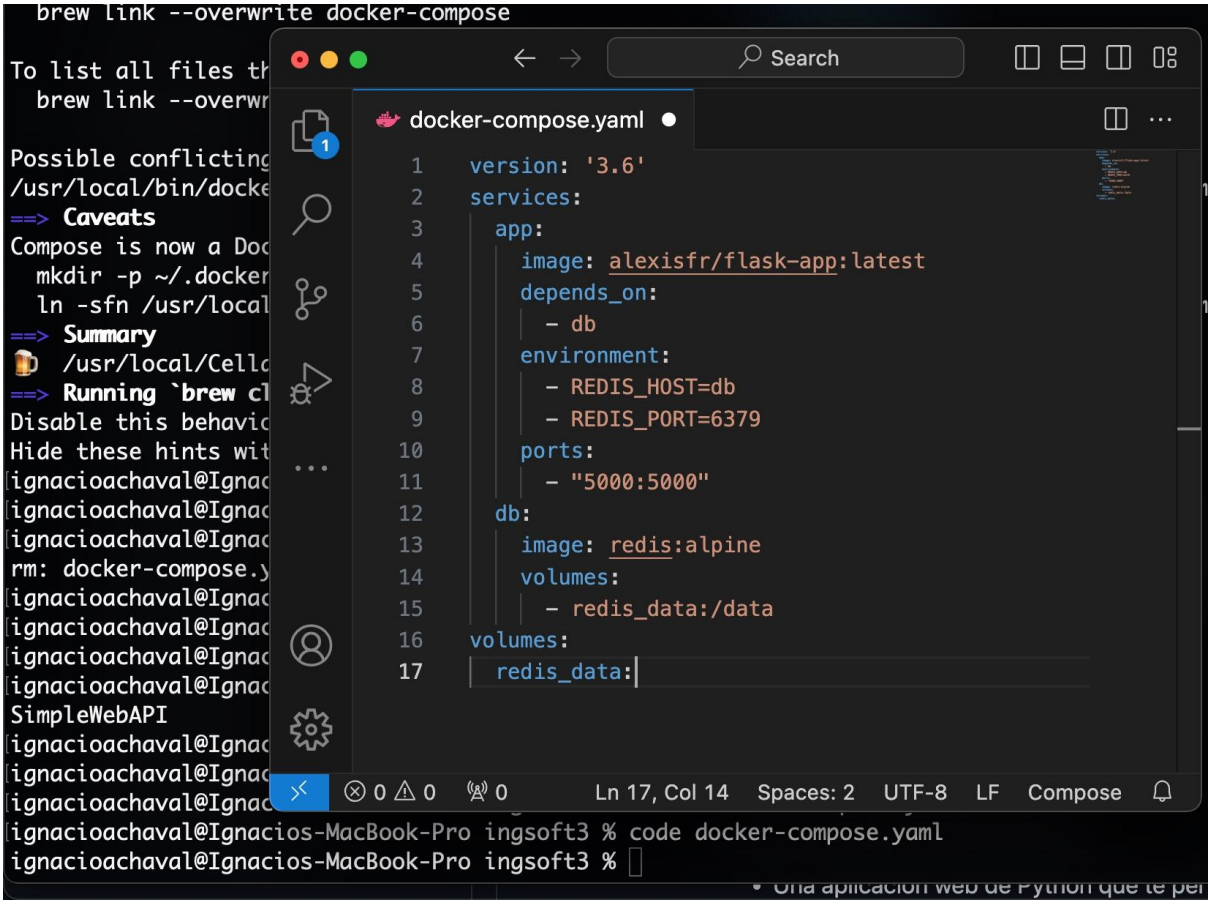
Se puede volver a levantar un nuevo contenedor de Redis después de eliminar el contenedor anterior, pero hay que tener en cuenta que cualquier dato que estaba almacenado en el contenedor original se perdió.

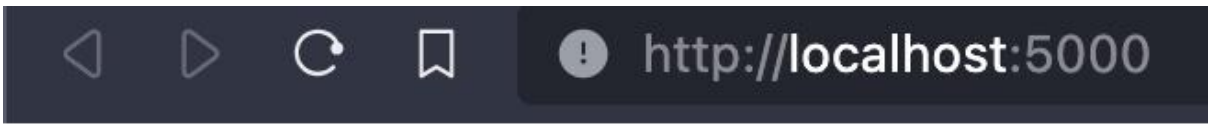
Para evitar perder la cuenta de las visitas cuando se elimina o reinicia el contenedor de Redis, es importante implementar una solución que permita la persistencia de los datos de Redis. Aquí tienes algunas estrategias que se pueden considerar:

- Persistencia de Datos en un Volumen Docker
- Réplicas de Redis
- Almacenamiento Externo de Redis

```
zsh: parse error near `}'
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker rm -f db

db
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker rm -f web

web
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker network rm mybridge

mybridge
```

3)





Hello from Redis! I have been seen 1 times.

```
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED          STATUS          PORTS                                       NAMES
d199291231d6   alexisfr/flask-app:latest  "python /app.py"         48 seconds ago   Up 46 seconds   0.0.0.0:5000->5000/tcp                      ingsoft3_app_1
55e36b44a19b   redis:alpine               "docker-entrypoint.s…"   49 seconds ago   Up 48 seconds   6379/tcp                                    ingsoft3_db_1
3ff5065fcd5b   postgres:9.4               "docker-entrypoint.s…"   52 minutes ago   Up 52 minutes   0.0.0.0:5432->5432/tcp                      my-postgres
f30cb9e70d74   350c60fdbbaa               "dotnet SimpleWebAPI…"   2 hours ago      Up 2 hours      443/tcp, 5254/tcp, 0.0.0.0:8080->80/tcp     awesome_ramanujan
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker network ls
NETWORK ID     NAME               DRIVER    SCOPE
a2bbc2300d01   bridge             bridge    local
d8a49b5281dc   host               host      local
a77dfa4ef725   ingsoft3_default   bridge    local
d800a91ba855   none               null      local
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 % docker volume ls
DRIVER    VOLUME NAME
local     a37e84dc7b1fdfa23f5ca1a08a050bdad9d045cf371a777872cd88a9057a487d
local     d837fe460c6d6ef3374d23766ea5744d03271310f933e1fc7e6fb820d9a3df3c
local     ingsoft3_redis_data
local     jenkins_home
ignacioachaval@Ignacios-MacBook-Pro ingsoft3 %
```

Docker compose hace por nosotros lo siguiente:

Versión de Compose:

version: '3.6' especifica la versión de Docker Compose que se utiliza. En este caso, se utiliza la versión 3.6.
Definición de Servicios:

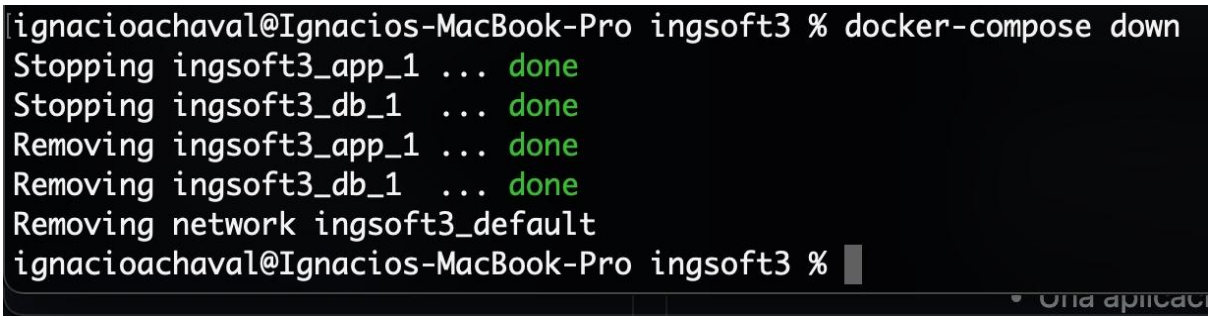
services:: Aquí comienza la definición de los servicios.
Servicio de la Aplicación Web (app):

image: alexisfr/flask-app:latest especifica la imagen que se utilizará para crear el contenedor de la aplicación web. En este caso, se utiliza la imagen alexisfr/flask-app con la etiqueta latest.
depends_on:: Indica que el servicio de la aplicación web depende del servicio de la base de datos Redis (db). Esto asegura que el servicio de la aplicación web se inicie después de que el servicio de Redis esté en funcionamiento.
environment:: Define las variables de entorno que se pasan al contenedor de la aplicación web. Se configura la dirección del host y el puerto de Redis a través de las variables REDIS_HOST y REDIS_PORT.
ports:: Mapea el puerto 5000 del host al puerto 5000 del contenedor, lo que permite acceder a la aplicación web a través del puerto 5000 del host.
Servicio de la Base de Datos Redis (db):

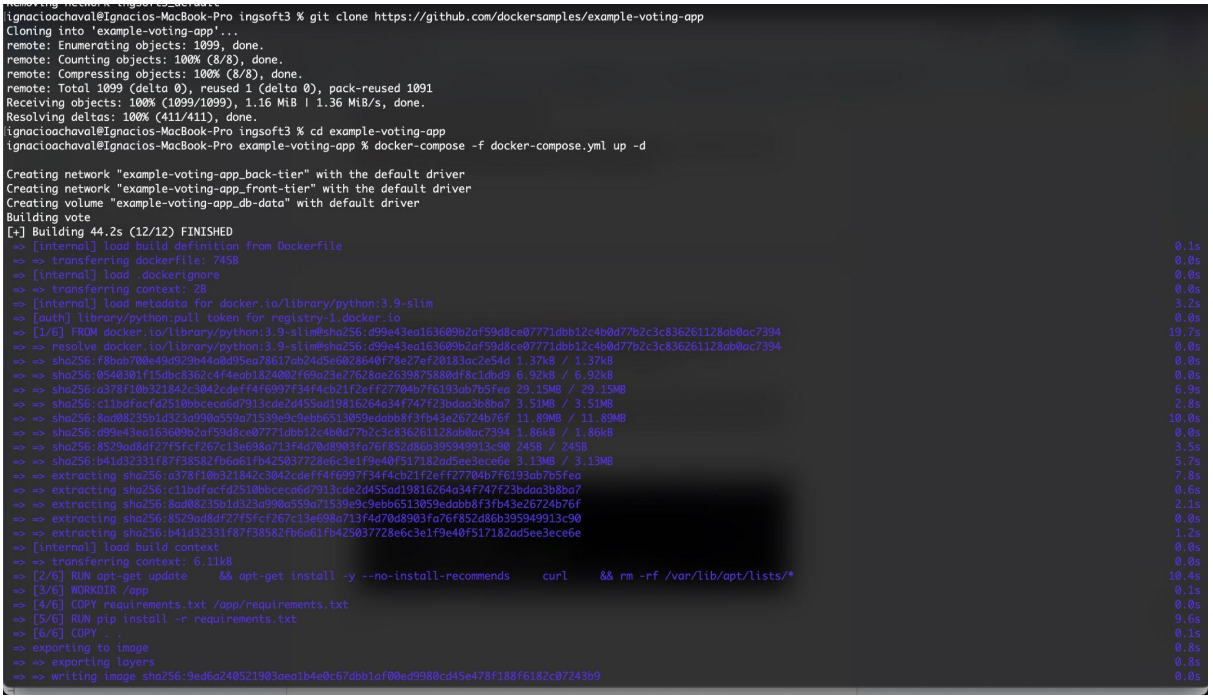
image: redis:alpine especifica la imagen que se utilizará para crear el contenedor de Redis. En este caso, se utiliza la imagen de Redis con la etiqueta alpine.

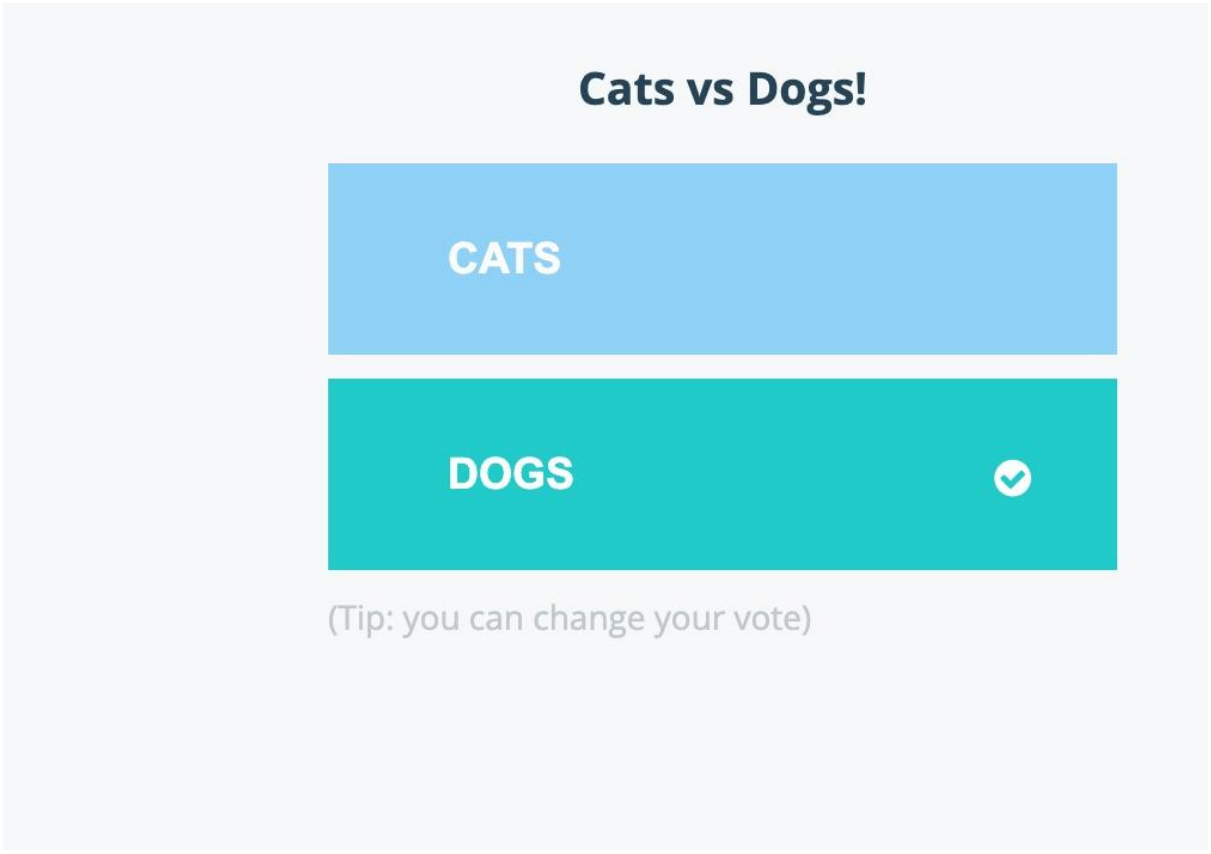
volumes:: Define un volumen llamado redis_data que se utiliza para almacenar los datos de Redis. Esto proporciona persistencia de datos, de modo que los datos de Redis se mantienen incluso si el contenedor de Redis se reinicia o se elimina.
Volúmenes:

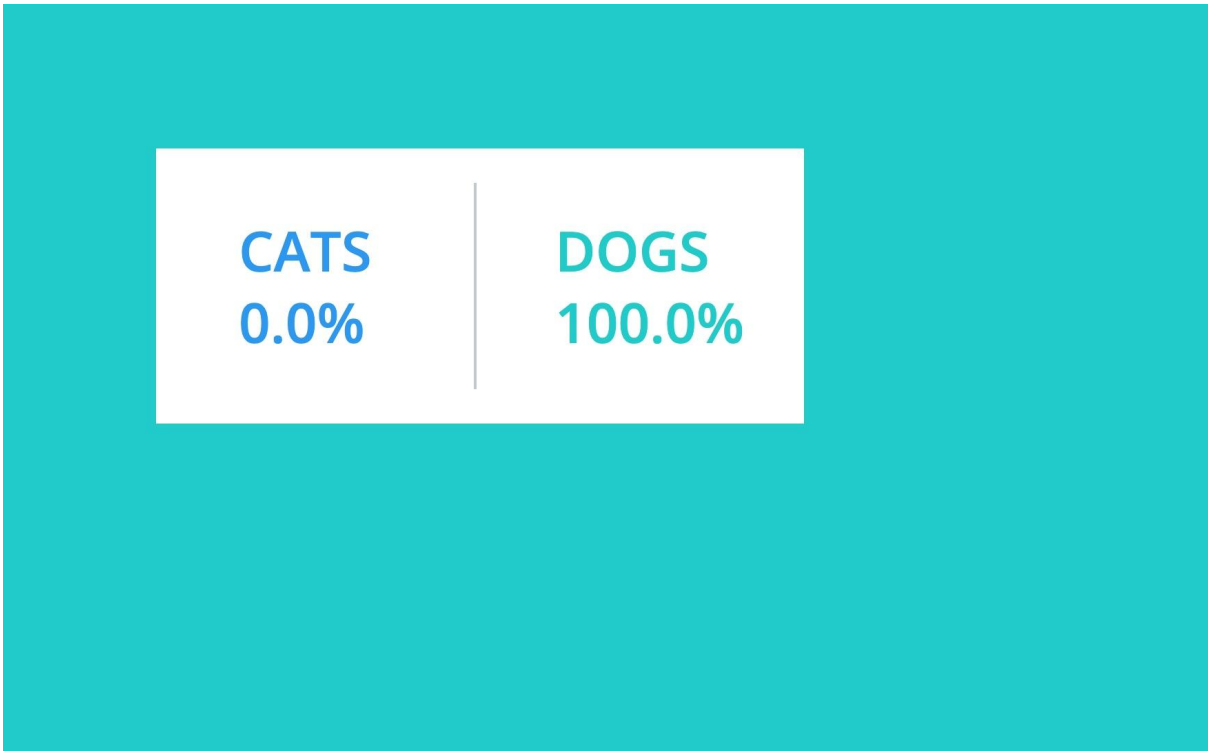
volumes:: Define un volumen llamado redis_data que se utiliza para almacenar los datos de Redis. Este volumen se utiliza en el servicio de Redis (db) para mantener la persistencia de datos.



4)







El sistema está configurado mediante el uso de Docker Compose y se compone de dos servicios: una aplicación web y una base de datos Redis.

Este archivo de Docker Compose orquesta dos servicios: la aplicación web y la base de datos Redis. La aplicación web depende de Redis para su funcionamiento, y ambos servicios utilizan un volumen llamado redis_data para garantizar que los datos de Redis se mantengan de forma persistente. El mapeo de puertos permite que la aplicación web sea accesible a través del puerto 5000 en el host.

5)

Users > ignacioachaval > workspace > src > github.com > ingsoft3 > docker-compose.yaml

```yaml
version: '3.6'
services:
  app:
    image: alexisfr/flask-app:latest
    depends_on:
      - db
    environment:
      - REDIS_HOST=db
      - REDIS_PORT=6379
    ports:
      - "5000:5000"
  db:
    image: redis:alpine
    volumes:
      - redis_data:/data
    ports:
      - "6379:6379"  # Agregar el mapeo de puerto para Redis
  postgres:
    image: postgres:alpine
    environment:
      POSTGRES_USER: your_username
      POSTGRES_PASSWORD: your_password
      POSTGRES_DB: your_database
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"  # Agregar el mapeo de puerto para PostgreSQL
volumes:
  redis_data:
  postgres_data:

```

Función get_redis():

Esta función se utiliza para obtener una instancia de la conexión a Redis. Si no existe una conexión en el contexto actual (objeto g), crea una nueva instancia de Redis y la almacena en g.redis.
Ruta Raíz ("/"):

La función hello() se encarga de manejar las solicitudes a la ruta raíz. Aquí es donde los usuarios pueden votar y se registran los votos en Redis.

Se obtiene el identificador del votante (almacenado en una cookie) o se genera uno nuevo si es la primera visita del usuario.

Cuando se envía un formulario de votación (método POST), se registra el voto en Redis. El voto se obtiene del formulario y se registra en la lista votes en Redis junto con el identificador del votante.

Se renderiza la plantilla HTML (index.html) que muestra las opciones de votación, el resultado actual y la información del host.

Finalmente, se configura una cookie con el identificador del votante para identificar al usuario en futuras visitas.

Configuración de Conexiones:

En el método Main, se establecen las conexiones tanto a la base de datos PostgreSQL (Postgres) como a la cola Redis.
La conexión a PostgreSQL se establece utilizando la biblioteca Npgsql, y la conexión a Redis se establece utilizando la biblioteca StackExchange.Redis.
Loop Infinito:

El programa se ejecuta en un bucle infinito con while (true). Esto permite que la aplicación esté en constante espera de nuevos votos en la cola Redis.
Procesamiento de Votos:

En cada iteración del bucle, la aplicación realiza varias acciones:
Consulta la cola Redis para obtener un voto en formato JSON con redis.ListLeftPopAsync("votes").Result. Este método retira y devuelve el primer elemento de la lista de votos en Redis.
Si se recibe un voto (el resultado no es nulo), se procesa el voto.
El voto se deserializa desde JSON a un objeto anónimo utilizando JsonConvert.DeserializeAnonymousType. Luego, se registra el voto en la base de datos PostgreSQL mediante la función UpdateVote.
Si no se recibe un voto (el resultado es nulo), se ejecuta una consulta de "mantenimiento" en PostgreSQL utilizando keepAliveCommand.ExecuteNonQuery(). Esto ayuda a mantener activa la conexión a la base de datos.
Manejo de Conexiones Redis y PostgreSQL:

Se incluye una lógica de manejo de conexiones para Redis y PostgreSQL. Si una conexión se cae o no está disponible, se intenta reconectar. Esto asegura que la aplicación pueda seguir funcionando incluso si uno de los servicios (Redis o PostgreSQL) se encuentra temporalmente no disponible.
Creación de Tabla en PostgreSQL:

Antes de registrar votos en PostgreSQL, se verifica que exista una tabla llamada votes. Si la tabla no existe, se crea mediante una consulta SQL.
Métodos Auxiliares:

Se incluyen varios métodos auxiliares, como OpenDbConnection para abrir una conexión a PostgreSQL y OpenRedisConnection para abrir una conexión a Redis. También hay un método UpdateVote para insertar o actualizar votos en PostgreSQL.
Manejo de Excepciones:

Se incluye un manejo de excepciones para garantizar que la aplicación siga funcionando incluso si se producen errores.
En resumen, este código del worker es una aplicación que se ejecuta continuamente, esperando nuevos votos en una cola Redis. Cuando se reciben votos, los registra en una base de datos PostgreSQL. También incluye lógica para mantener activas las conexiones a Redis y PostgreSQL y para crear la tabla en PostgreSQL si aún no existe. El código garantiza la integridad y persistencia de los datos de votación.

El código en C# que se muestra es el programa del trabajador (worker) de una aplicación de votación. El trabajador se encarga de procesar los votos enviados a través de Redis y almacenarlos en una base de datos PostgreSQL. A continuación, se explica cómo funciona este programa:

1. **Configuración Inicial**:
   - El programa comienza estableciendo conexiones con la base de datos PostgreSQL (`pgsql`) y Redis (`redisConn`) utilizando las cadenas de conexión proporcionadas.
   - Se crea un objeto `definition` para definir la estructura de los datos JSON que se obtienen de Redis. En este caso, se espera que los datos contengan dos campos: `vote` y `voter_id`.

2. **Bucle Principal**:
   - El programa entra en un bucle infinito (`while (true)`) para procesar continuamente los votos.
   - Para evitar una alta carga en la CPU, el programa se duerme durante 100 milisegundos en cada iteración del bucle (`Thread.Sleep(100)`).

3. **Reconexión Redis**:
   - Se verifica si la conexión a Redis está abierta. Si no está abierta, el programa intenta reconectar y restablecer la conexión a Redis.

4. **Procesamiento de Votos**:
   - Se obtiene un voto de la cola de votos en Redis utilizando `redis.ListLeftPopAsync("votes").Result`. Esto elimina el voto de la cola de Redis.
   - Si se obtiene un voto válido desde Redis, se deserializa el JSON y se almacena en la variable `vote`.
   - Se registra en la consola un mensaje que indica que se está procesando un voto, incluyendo el voto y el votante.

5. **Reconexión PostgreSQL**:
   - Se verifica si la conexión a la base de datos PostgreSQL está abierta. Si no está abierta, el programa intenta reconectar y restablecer la conexión a la base de datos.

6. **Actualización de Votos en la Base de Datos**:
   - Si la conexión a PostgreSQL está abierta, se ejecuta una consulta SQL para insertar o actualizar el voto en la base de datos. Si el votante ya ha votado, se actualiza su voto; de lo contrario, se inserta un nuevo voto.

7. **Manejo de Excepciones**:
   - Se manejan excepciones en caso de que ocurra un error en la conexión a la base de datos o Redis.

8. **Método `OpenDbConnection`**:
   - Este método se utiliza para abrir una conexión a la base de datos PostgreSQL. Realiza un bucle hasta que la conexión sea exitosa. También se crea una tabla llamada "votes" si no existe.

9. **Método `OpenRedisConnection`**:
   - Este método se utiliza para abrir una conexión a Redis. Realiza un bucle hasta que la conexión sea exitosa.

10. **Método `GetIp`**:
    - Este método se utiliza para obtener la dirección IP a partir de un nombre de host. En este caso, se utiliza para resolver la dirección IP de Redis.

11. **Método `UpdateVote`**:
    - Este método se encarga de actualizar la base de datos con un nuevo voto. Si el votante ya ha votado, se actualiza su voto; de lo contrario, se inserta un nuevo voto en la tabla "votes" de PostgreSQL.

En resumen, este programa en C# procesa continuamente votos enviados desde una cola Redis, los almacena en una base de datos PostgreSQL y se encarga de la reconexión a Redis y PostgreSQL en caso de pérdida de conexión. Es parte de una aplicación de votación que permite a los usuarios emitir votos y mantener un registro de los mismos.

Documento de Arquitectura del Sistema de Votación

Introducción

Este documento presenta una descripción de la arquitectura del sistema de votación, que permite a los usuarios votar por una de dos opciones, "Cats" o "Dogs". El sistema está compuesto por varios componentes que trabajan juntos para permitir la votación, el procesamiento de votos y el almacenamiento de resultados.

Arquitectura General

La arquitectura del sistema de votación se compone de los siguientes componentes:

1. **Aplicación Web (Frontend)**:
   - La interfaz de usuario a través de la cual los usuarios pueden votar por sus opciones preferidas.
   - Escrita en HTML, CSS y JavaScript.
   - Utiliza el framework Flask para el desarrollo de la aplicación web.

2. **Aplicación Worker (Backend)**:
   - Un trabajador que procesa los votos recibidos desde la aplicación web y los almacena en una base de datos.
   - Escrito en C#.
   - Utiliza la base de datos PostgreSQL para almacenar los votos y Redis para la gestión de la cola de votos.

3. **Base de Datos PostgreSQL**:
   - Almacena los votos de los usuarios en una tabla llamada "votes".
   - Se comunica con la aplicación worker para registrar y actualizar los votos.

4. **Redis**:
   - Utilizado como una cola de mensajes para gestionar los votos enviados por los usuarios.
   - El trabajador procesa los votos de la cola Redis y los almacena en PostgreSQL.

Interacciones entre Componentes

A continuación, se describen las interacciones clave entre los componentes del sistema:

Interacción de Votación

1. Un usuario interactúa con la Aplicación Web a través de su navegador y emite un voto seleccionando una de las dos opciones: "Cats" o "Dogs".

2. La Aplicación Web envía el voto del usuario a la cola Redis utilizando una solicitud HTTP.

3. El Aplicación Worker monitorea constantemente la cola Redis para nuevos votos. Cuando detecta un voto en la cola, lo procesa.

4. El Aplicación Worker extrae el voto de la cola Redis y lo registra en la base de datos PostgreSQL. Si el usuario ya ha votado, se actualiza el voto existente.

Interacción de Almacenamiento

1. La Aplicación Web se comunica con PostgreSQL para obtener los resultados de la votación y mostrarlos en la interfaz de usuario.

2. El Aplicación Worker también se comunica con PostgreSQL para registrar y actualizar los votos.

Comunicación de Servicios

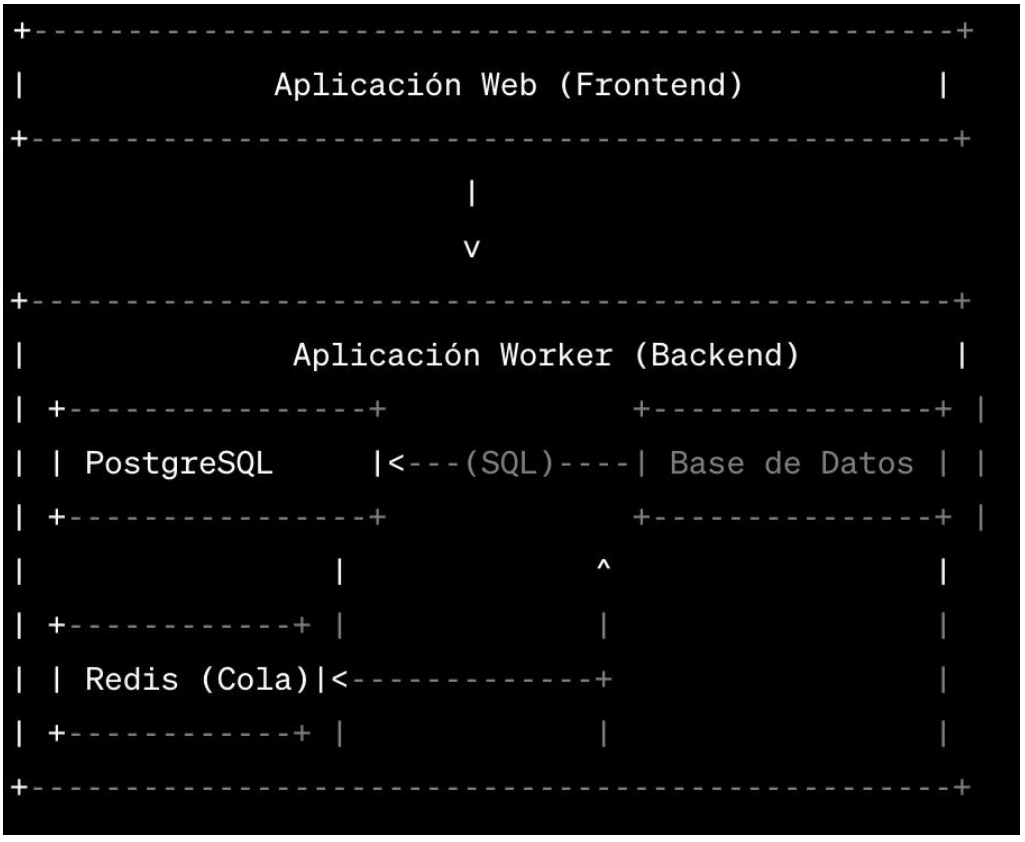
- Tanto la Aplicación Web como el Aplicación Worker se comunican con la base de datos PostgreSQL utilizando consultas SQL.

- El Aplicación Worker se comunica con la cola Redis para obtener y procesar votos.

Diagramas

A continuación se presentan algunos diagramas que ilustran la arquitectura y las interacciones clave:

Diagrama de Bloques de la Arquitectura



Conclusiones

La arquitectura del sistema de votación permite a los usuarios votar por sus opciones preferidas y garantiza que los votos se procesen y almacenen de manera efectiva. La Aplicación Web proporciona una interfaz amigable para los usuarios, mientras que el Aplicación Worker se encarga de la gestión y procesamiento de los votos. La base de datos PostgreSQL y Redis se utilizan para el almacenamiento de datos y la gestión de la cola de votos, respectivamente. En conjunto, estos component