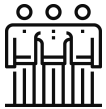
Microservicios

Mario Alberto Cruz González

Academy Code of Conduct

Seamos respetuosos, no existen preguntas malas



Seamos pacientes

Cuidemos nuestro lenguaje

Objetivos de la sesión

Al final de la sesión seremos capaces de:

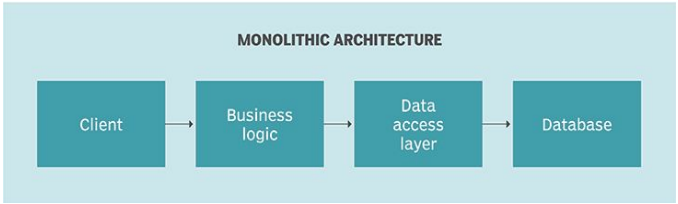
- Comprender que es un microservicio, cómo se diferencía de un monolito y que ventajas tiene.
- Cual es la arquitectura de microservicios
- Qué es docker y cómo se utiliza en el desarrollo de aplicaciones y microservicios.
- Qué es docker compose y como se usa en el desarrollo.

Tabla de contenido

Microservicios vs Monolitos

Monolito

Una aplicación cuyos componentes son una unidad acoplada, en la cual un componente depende directamente de los otros para funcionar.



Microservicios

Un sistema en el que diferentes aplicaciones o servicios cumplen con tareas específicas, permitiendo así que las aplicaciones que se presentan a los usuarios finales sean módulos pequeños y fáciles de manejar.

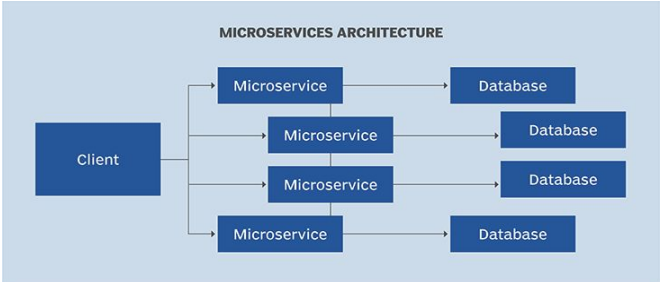


Tabla de contenido

Arquitectura de microservicios

Microservicios

Una arquitectura de microservicios divide una aplicación en una serie de servicios implementables de forma independiente que se comunican a través de API.

- Varios servicios de componentes
- Muy fáciles de mantener y de probar
- Pertenecen a equipos pequeño
- Se organizan en torno a capacidades empresariales
- Infraestructura automatizada

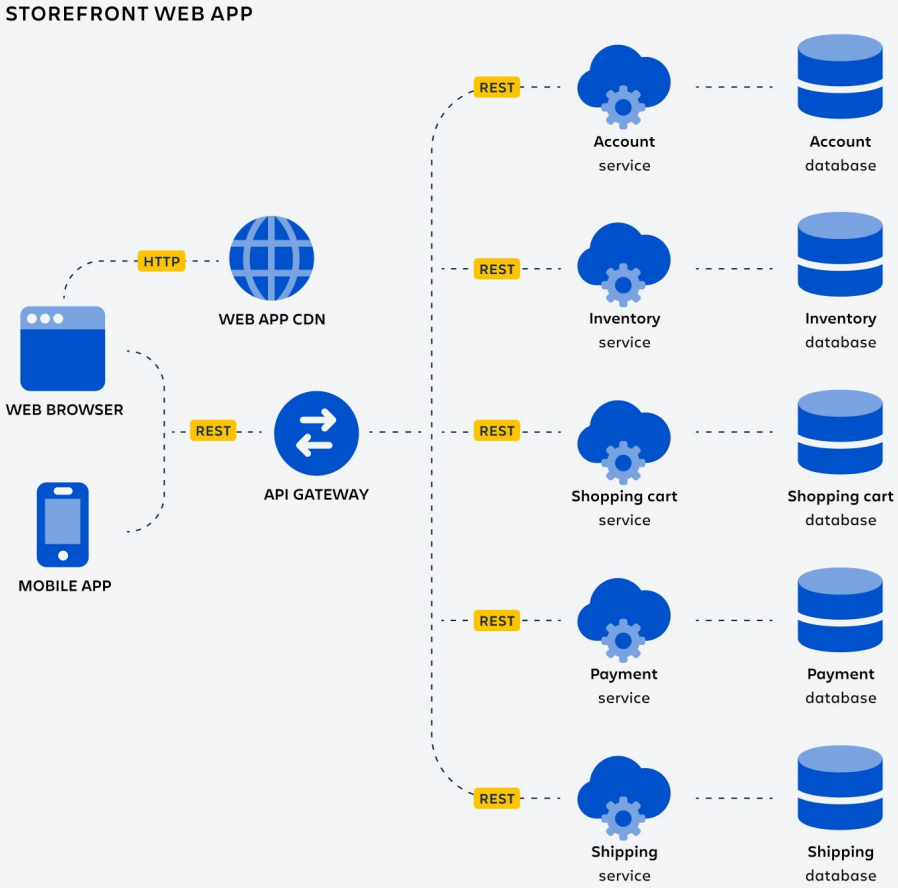
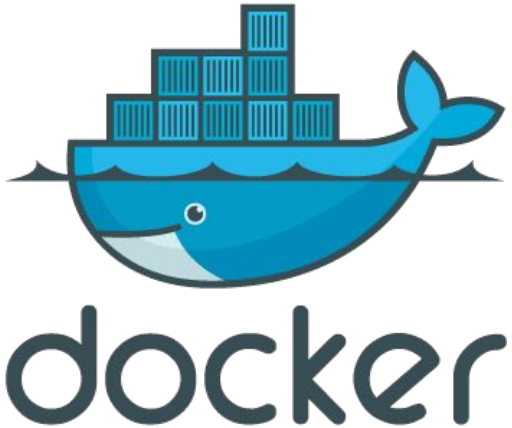


Tabla de contenido

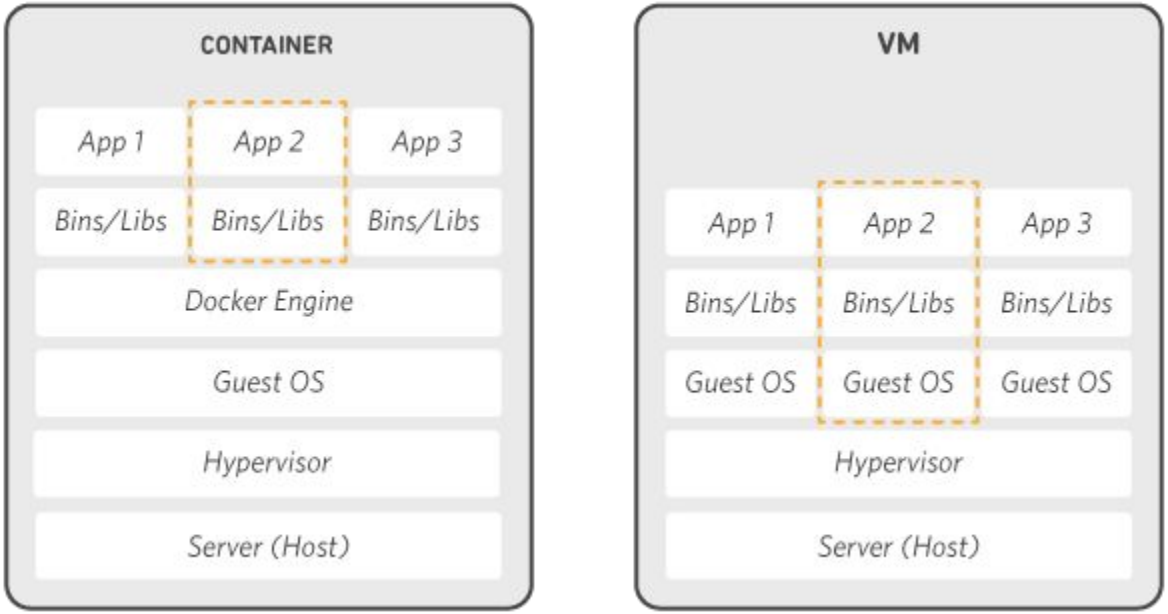
Docker

Docker

Docker es una plataforma comercial de contenedorización y un tiempo de ejecución de contenedores que ayuda a los desarrolladores a crear, implementar y ejecutar contenedores.

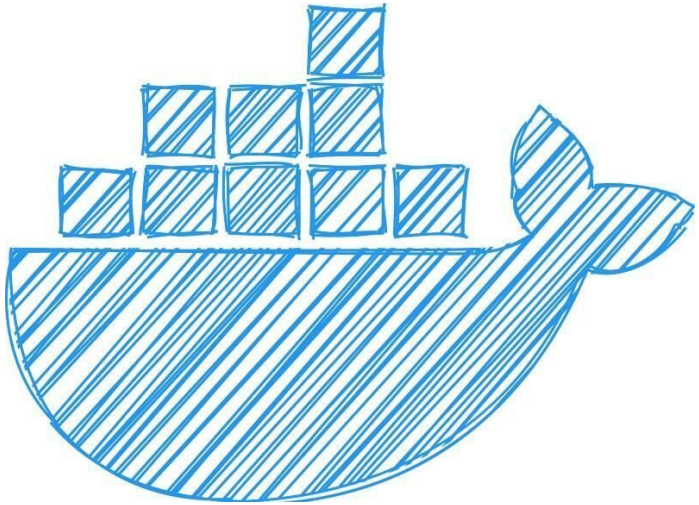


Contenerización vs Virtualización



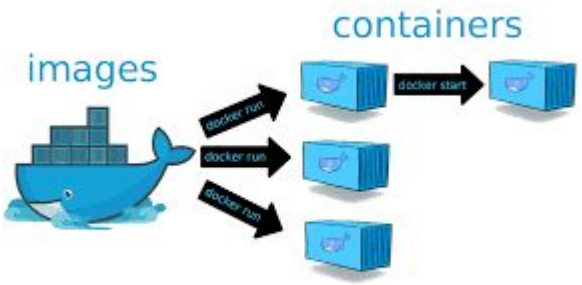
Imagen

Una imagen es el “plano” que contiene las instrucciones con base en las cuales se ejecutará un contenedor de docker. Provee una manera conveniente de empaquetar aplicaciones y ambientes preconfigurados los cuales se pueden usar de manera privada o pública, compartiendo dichas imágenes con otros usuarios de docker a través de repositorios como Docker Hub.



Contenedor

Un contenedor Docker es un formato que empaqueta todo el código y las dependencias de una aplicación en un formato estándar que permite su ejecución rápida y fiable en entornos informáticos. Es un proceso del sistema operativo, como cualquier otro, pero tiene un sistema de archivo, conexión de red y árbol de procesos que está aislado del resto del sistema operativo huésped.



Hands ON!

- Instalar docker
  - Mac:
    - brew install colima
    - brew install docker
    - brew install kubectl
    - brew install docker-compose
  - Windows:
    - Install docker desktop or rancher
      https://www.docker.com/products/docker-desktop/
      https://docs.rancherdesktop.io/getting-started/installation/
  - Linux:
    - sudo apt-get update
    - sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
    - sudo docker run hello-world

- Creamos una carpeta en la que almacenaremos el código y configuración para las imagenes docker que usaremos.

Como codigo usaremos el siguiente script en python, cuyo nombre deberá ser app.py

```
import time


from flask import Flask


app = Flask(__name__)



@app.route('/')
def hello():
    return 'Hello World FROM DOCKER!'
```

Dockerfile

Es un archivo de texto que contiene las instrucciones con las cuales Docker va a crear una imagen. Adicionalmente agregaremos un archivo llamado requirements.txt, el cual contiene los requerimientos de la aplicación que serán instalados en el contenedor. (Este archivo es especifico para ciertas aplicaciones python, no siempre será requerido).

requirements.txt

```
flask
```

Dockerfile

```
# syntax=docker/dockerfile:1
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

Construir la imagen:

```
docker build --tag python-docker .
```

Ver imagenes locales:

```
docker images
```

Etiquetar imagenes:

```
docker tag python-docker:latest python-docker:v1.0.0
```

Importante: Etiquetar una imagen no crea una nueva imagen, solo nos permite referenciar a la misma imagen con un nombre distinto, al volver a lista las imágenes podremos ver que el id de las imágenes en ambas etiquetas es el mismo.

Remover etiqueta:

```
docker rmi python-docker:v1.0.0
```

Instanciamos un contenedor de la imagen:

```
docker run python-docker
```

Intentamos llamar a la aplicación:

```
curl --request GET \

--url http://localhost:8080/
```

Exponemos el puerto en el cual la aplicación se está ejecutando:

```
docker run --publish 8000:5000 python-docker
```

Correr la aplicación en modo detached:

```
docker run -d -p 8000:5000 python-docker
```

Listar contenedores:

```
docker ps
```

Detener contenedor:

```
docker stop trusting_beaver
```

Listar todos los contenedores

```
docker ps -a
```

Reiniciar contenedor

```
docker restart trusting_beaver
```

Eliminar contenedores

```
docker rm trusting_beaver modest_khayyam lucid_greider
```

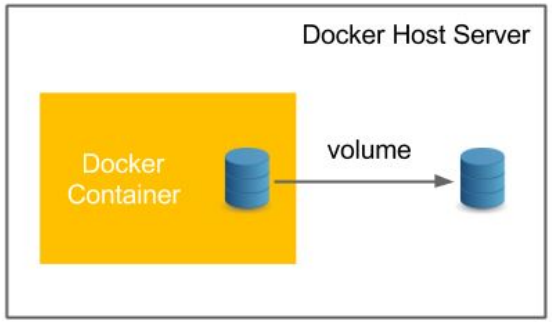
Correr contenedor con nombre (y autolimpieza)

```
docker run --rm -d -p 8000:5000 --name python-server python-docker
```

Volúmenes

Los datos usados en un contenedor se almacenan en volúmenes, una entidad persistente internamente manejada por docker y administrada por el usuario a través de comandos docker. Los datos de aplicación se recomienda sean almacenados en volúmenes ya que son independientes del ciclo de vida del contenedor.

```
docker run -d --name geekflare  -v geekvolume:/app nginx:latest
```



Red

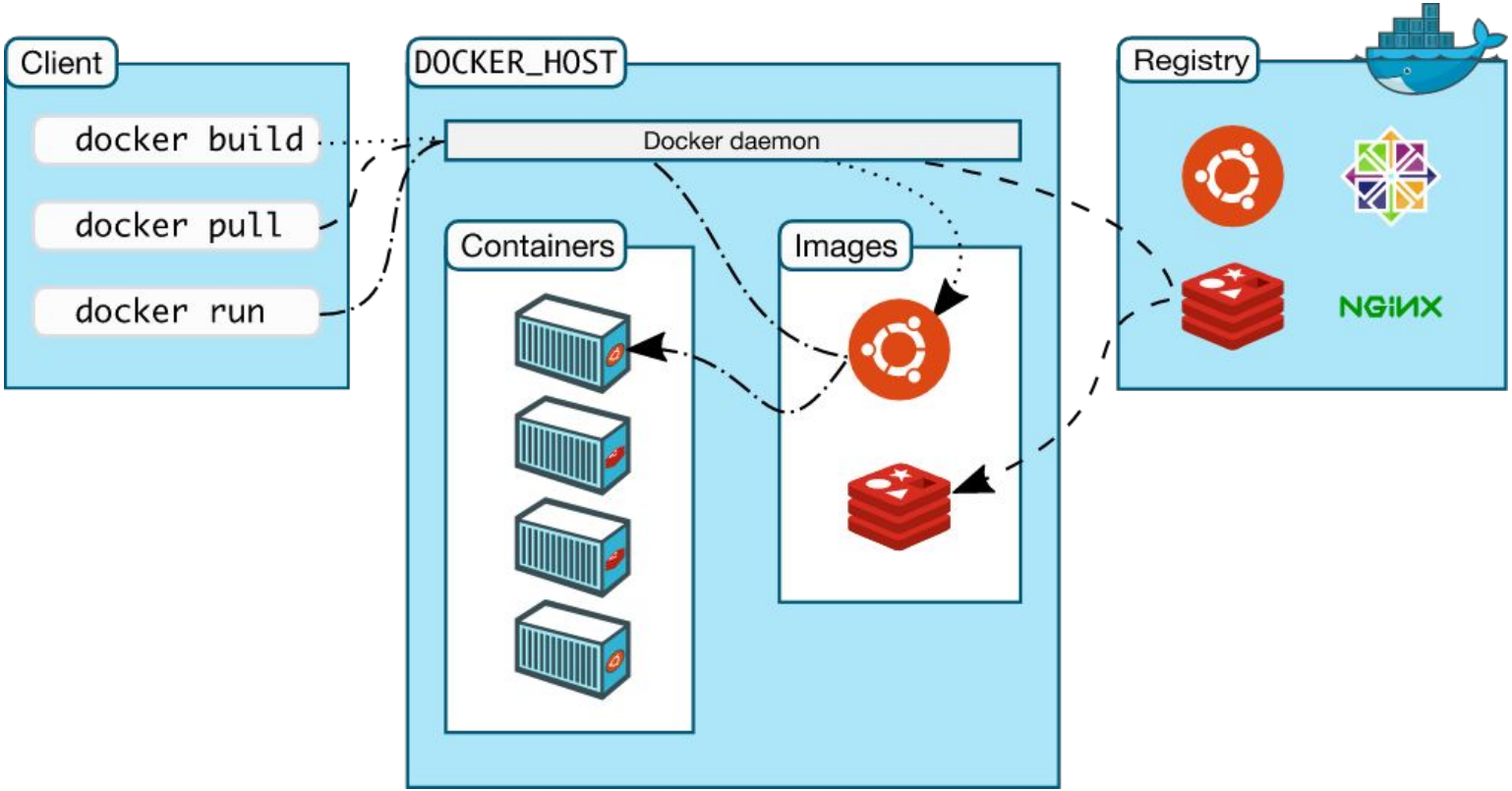
Como se mencionó anteriormente los contenedores tienen su propia red aislada del resto del host, esta red tiene la posibilidad de tener alguno de los siguientes drivers:

- **Bridge.-** Default, usada para contenedores *standalone,* por ejemplo que solo se comunican entre ellos
- **Host.-** Remueve el aislamiento con el host
- **Overlay.-** Permite la comunicación entre contenedores en otros hosts
- **None.-** Sin red
- **macvlan.-** Asigna una dirección MAC virtual que permite que los contenedores sean vistos como direcciones físicas, útil en migraciones de VM a contenedor

Registros/Repositorios

Es la donde las imágenes se almacenan para ser recuperadas e instanciadas como contenedores por docker. Dichos repositorios pueden ser públicos o privados, siendo docker hub el más popular entre los públicos y el default para docker.

Arquitectura Docker



Motor Docker

Es el núcleo de docker, se instala en el sistema anfitrion, en un modelo **cliente-servidor** y está compuesto por:

- **Servidor.-** Es un demonio llamado dockered que se encarga de la creación y gestión de los contenedores y sus componentes.
- **API rest.-** Se encarga de decirle al demonio de docker qué hacer
- **CLI.-** Interfaz de línea de comandos para ejecutar operaciones docker
- **Cliente.-** La forma en que el usuario se comunica con docker

Docker fuera de Linux

RANCHER®

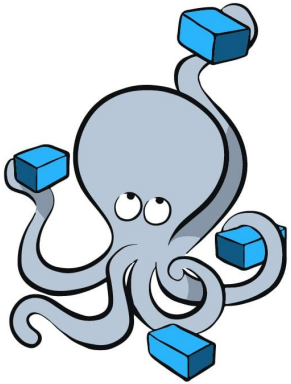
lima

Tabla de contenido

Docker Compose

Docker Compose

Docker compose es una herramienta para definir y ejecutar aplicaciones docker de múltiples contenedores. Con compose definiremos los servicios a ser ejecutados en un solo archivo .yaml a partir del cual, con un solo comando, tendremos todo el ambiente requerido por la aplicación corriendo aisladamente



docker
Compose

Usos comunes

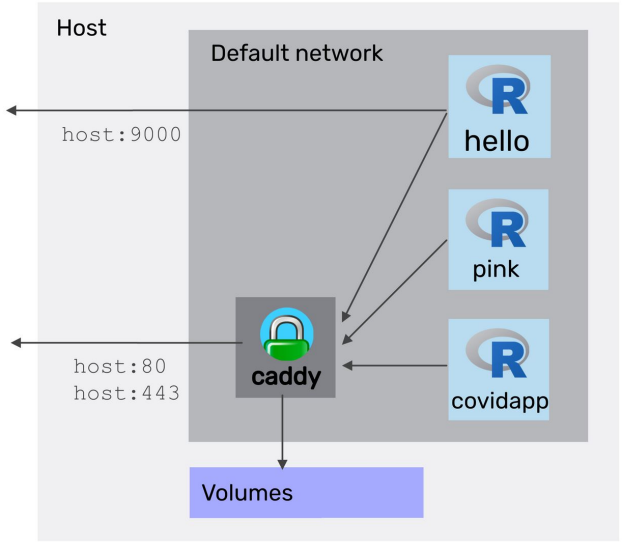
- Ambientes de desarrollo
- Ambientes para pruebas automatizadas
- Liberaciones a un solo host

```
version: "3.7"

services:
  hello:
    image: hello
  pink:
    build: ./pink
  covidapp:
    image: covidapp
  caddy:
    image: caddy

volumes:
```

docker-compose.yml



Manos a la obra

Redefinimos la sencilla aplicación que hicimos en el ejercicio anterior.

```
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

Definimos su dockerfile y requerimientos

```
requirements.txt
flask
redis
```

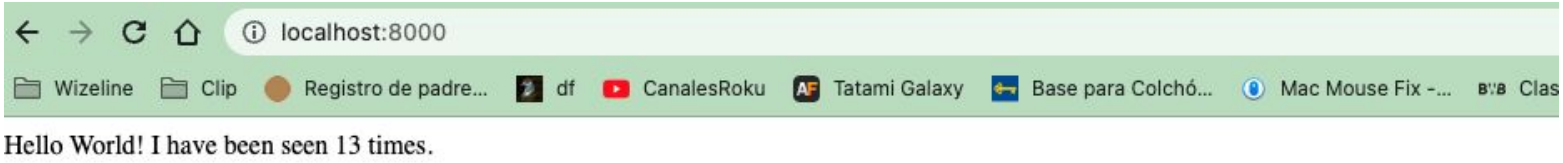
```
# syntax=docker/dockerfile:1
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

Definimos el archivo docker-compose.yml

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

Ejecutamos y probamos la aplicación definida

```
docker-compose up
```



Editamos el archivo compose para tener un volumen montado al contenedor de la aplicacion web.

```yaml
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

Este cambio nos permitirá editar el código de esta pequeña aplicación y ver dichos cambios sin necesidad de reconstruir el ambiente. Reconstruimos una vez más para aplicar el volumen:

```
docker compose up
```

Modificamos el archivo app.py, cambiemos el mensaje que responderá la aplicación y actualizamos algunas veces la aplicación, ya que los cambios pueden tomar un poco en verse reflejados:

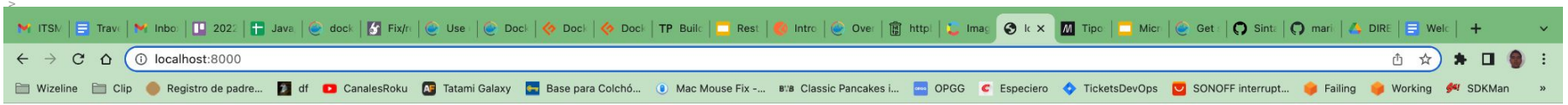
```
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World from DOCKER! I have been seen {}
times.\n'.format(count)
```



Hello World FROM DOCKER! I have been seen 44 times.

Comandos Docker adicionales:

Correr en modo deatached

docker-compose up -d

Servicios que están corriendo actualmente

docker-compose ps

Diferentes comandos en algun servicio

docker-compose run

En este ejemplo se ejecuta en el servicio web env, que listara las variables de ambiente

docker-compose run web env

Detener los servicios, por ejemplo, cuando se inicien en modo deatached

docker-compose stop

Detener completamente los servicios y remover la data en los volumenes:

docker compose down --volumes

Thank you