Contenedores: más que VMs - Docker

XII Edición Bootcamp DevOps & Cloud Computing Full Stack
Evaristo García Zambrana | 3 de agosto de 2025

Objetivo: Desplegar una aplicación en arquitectura de microservicios a través de Docker Compose.

ÍNDICE

Descripción general

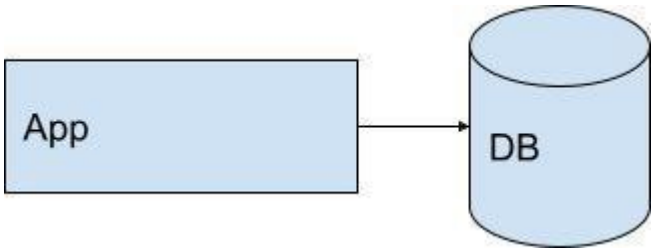
En esta práctica debemos implementar una aplicación consistente en un micro servicio que sea capaz de leer y escribir en una base de datos.

La aplicación tendrá que poderse desplegar con docker-compose.

El microservicio / aplicación y la base de datos son de tu elección.

Puedes basarte en el flask-counter que hemos estudiado durante el curso, desarrollar tu propia aplicación sencilla o basarte en alguna aplicación existente de tu elección (NodeJS, Java, Python, Go, …).

Puedes usar cualquier lenguaje de programación y cualquier framework.



Hitos

Crear repositorio GIT con la aplicación.

El repositorio se ha creado en mi perfil personal de GitHub: https://github.com/EvaristoGZ/microservicios-Docker-Compose

Crear un fichero README.md con todas las instrucciones necesarias para poder comprender el proyecto y ponerlo en funcionamiento.

Descripción de la aplicación.

Se trata de una aplicación desarrollada con Python 3.13 que registra con fecha y hora cada visita que se realice a la dirección URL que expone. Además, mostrará la fecha y hora de las últimas diez visitas.

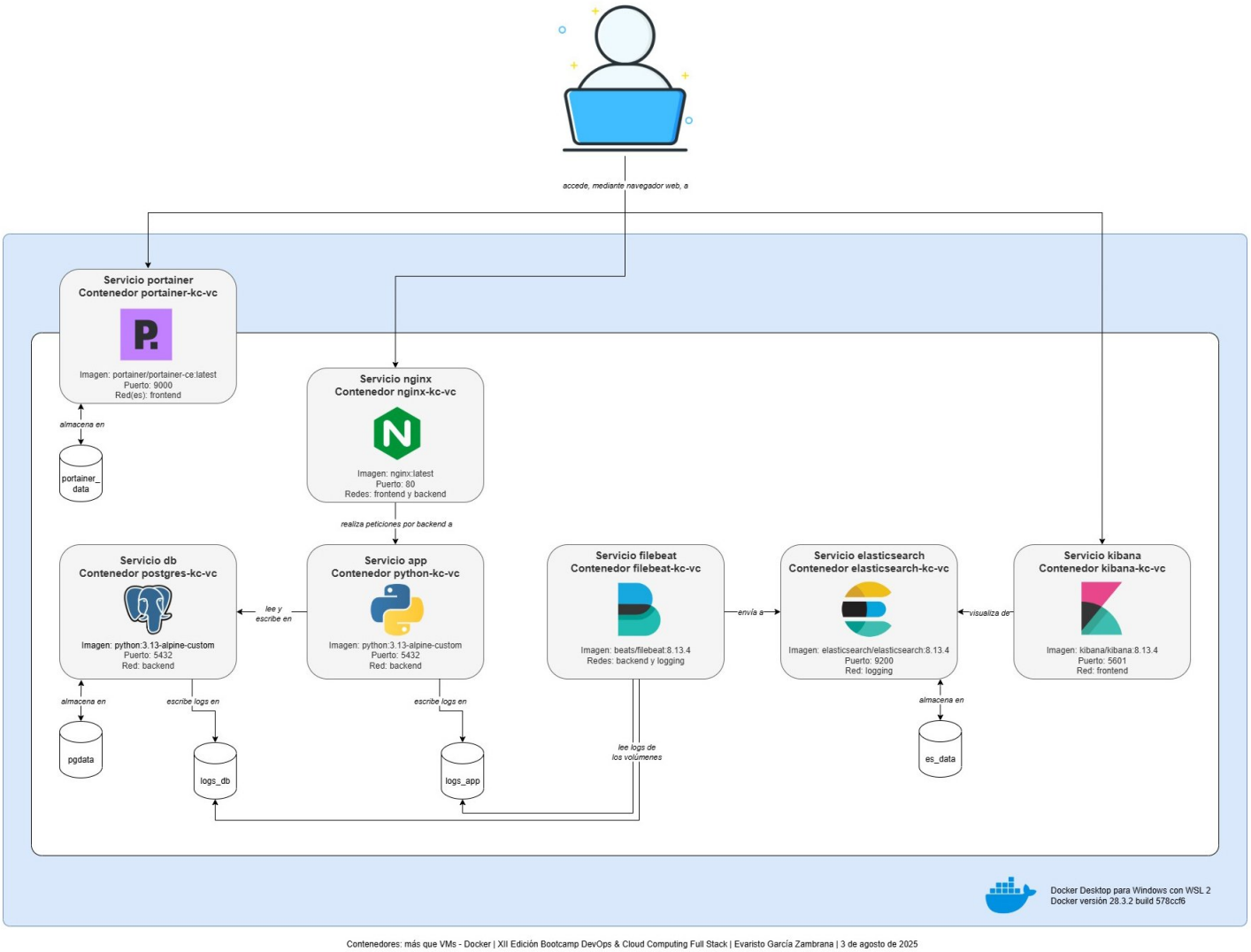
Este control de visitas se almacena en una base de datos PostgreSQL 15.

La aplicación correrá en un contenedor independiente, con una imagen compilada o construida a la vez que se levanta el docker-compose mientras que la base de datos usará la imagen oficial de Postgres con la etiqueta 15.

Los datos serán persistentes a través de un volumen llamado *pgdata*.

Funcionamiento de la aplicación.

Diagrama del resultado final de la ejecución de todos los pasos de esta práctica, quedando un total de siete servicios de Docker Compose.



Requisitos para hacerla funcionar.

Instrucciones para ejecutarla en local y verificar el funcionamiento correcto.

Los requisitos para la ejecución de esta aplicación son:

1. Tener instalado Git
2. Clonar el repositorio https://github.com/EvaristoGZ/microservicios-Docker-Compose
3. Tener instalado Docker Engine o Docker Desktop.
4. Tener instalado Docker Compose. Docker Desktop ya instalará, por defecto, Docker Compose.
5. Ubicarnos en el repositorio git.

Si se usa Windows con WSL 2 es posible que debamos establecer una variable para el directorio real del repositorio git que descargamos. Ejecutamos ubicados dentro de ese directorio: `export PWD=$(realpath .)`

También es necesario crear un fichero .env en el directorio. Este tiene el siguiente contenido:

docker-compose.yml

```
# Aprovisionamiento PostgreSQL
POSTGRES_USER=postgres
POSTGRES_PASSWORD=SUPER-PA$$WORD
POSTGRES_DB=kc-visit-counter

# Cadena conexión BBDD
DB_NAME=kc-visit-counter
DB_USER=kc-user
DB_PASSWORD=kc-PA$$WORD
DB_HOST=db
DB_PORT=5432

# Otros
TZ=Europe/Madrid
```

Posteriormente, basta con hacer un `docker compose up --build` en la primera ejecución para que el conjunto de servicios, contenedores, redes y volúmenes se creen y levanten.

Con `docker compose down` pararemos el Compose. Mientras que con `docker compose down -v` no solo lo pararemos, si no que eliminaremos también los volúmenes y redes creados en Docker mediante el fichero *docker-compose.yml* que contiene el repositorio.

Dockerfile que construya el / los contenedores necesarios:

Que compile, instale dependencias, etc.

En un principio, la lógica de la creación del esquema de la base de datos recaía en la propia aplicación Python, pero posteriormente se cambió la estructuración para que fuese el servicio *db* el encargado de aprovisionar lo relacionado a PostgreSQL, siendo necesario que la aplicación únicamente conecte y escriba en la base de datos las visitas.

Con motivo de simplificar, el servicio de base de datos *db* levanta con una imagen PostgreSQL, con tag 15.

La aplicación Python requería de algo más de complejidad, por lo que sí se generaba una imagen a través del Dockerfile ubicado en *app*.

Que la empaquete con los requisitos mínimos (usar Multistage)

Docker compose

Que permita ejecutar la aplicación completa en local, considerando la persistencia de datos y comunicaciones entre distintos contenedores.

Este es el *docker-compose.yml* que levanta el servicio de base de datos y la aplicación.

docker-compose.yml

```
services:
  db:
    image: postgresql:15
    container_name: postgres-kc-vc
    env_file: .env
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./postgresql/postgresql.conf:/etc/postgresql/postgresql.conf
    ports:
      - "5432:5432"

  app:
    build: ./app
    image: kc/python:3.13-bookworm-custom
    container_name: python-kc-vc
    environment:
```

```
      DB_HOST: db
      DB_PORT: 5432
      DB_NAME: kc-visit-counter
      DB_USER: kc-user
      DB_PASSWORD: kc-PA$$WORD
      TZ: Europe/Madrid
    depends_on:
      - db
    ports:
      - "5050:5000"

volumes:
  pgdata:
```

En el caso de la base de datos, se usa la imagen oficial de PostgreSQL con tag 15.
Se le indica que este servicio tiene un volumen asociado a la ruta */var/lib/postgresql/data*, el cual tiene como nombre "*pgdata*". Este volumen mantendrá los datos aunque el compose y los contenedores se apaguen y/o eliminen. Sólo se borrarán los datos cuando se elimine el volumen.

La creación del esquema de la base de datos es controlada a través del código de la aplicación, identificando si existe o no la base de datos.

En cuanto al servicio app, construye la imagen personalizada a través del *dockerfile* ubicado en el mismo directorio. Tiene un port binding del puerto 5000 del contenedor (dónde la aplicación Python con Flask levanta) hacia el puerto 5050 de la máquina anfitrión. Este servicio depende del servicio *db*, por lo que iniciará primero el servicio *db* antes que el de *app*. La imagen usada en un principio tiene como base Debian Bookworm, con la finalidad de que algunos de los puntos siguientes sean más completos como el análisis de vulnerabilidades y seguridad, o la revisión de capas y reducción del tamaño de imagen base.

Ambos contenedores tienen establecido el timezone Europe/Madrid, pues la aplicación registra la hora de cada visita.

Este es el *docker-compose.yml* base y con los requisitos de hasta ahora, que luego se mejorará con los siguientes enunciados y puntos extras dando como resultado el *docker-compose.yml* que permanece en el repositorio de GitHub.

Instrucciones para verificar el funcionamiento correcto.

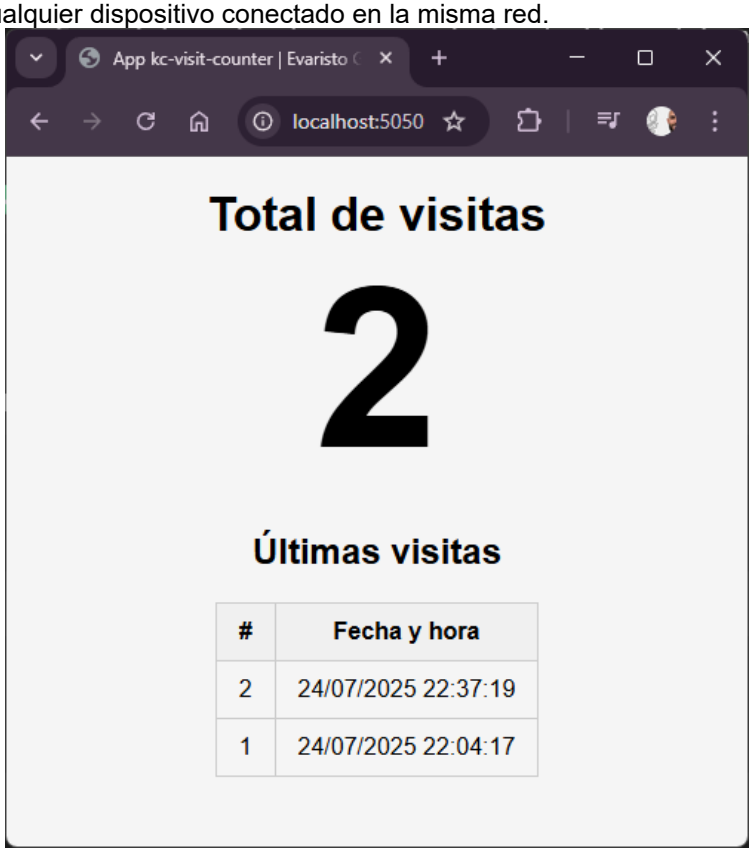
Tras tener instalado Docker y Docker Compose en nuestro SO, ubicados en el directorio ejecutamos:

```
docker compose up --build
```

que hará un pull de las distintas capas de las imágenes, hará el building o construcción de la imagen personalizada para el servicio app (cogiendo la configuración del *dockerfile*), creará la red *kc-visit-counter_default*, el volumen *kc-visit-counter_pgdata* y por último creará los dos contenedores.

```
egz@thinkpad:~/kc-visit-counter$ docker compose up --build
[+] Building 1.0s (22/22) FINISHED
 => [internal] load local bake definitions
 => => reading from stdin 662B
 => [app internal] load build definition from Dockerfile
 => => transferring dockerfile: 312B
 => [db internal] load build definition from Dockerfile
 => => transferring dockerfile: 341B
 => [app internal] load metadata for docker.io/library/python:3.13-bookworm
 => [db internal] load metadata for docker.io/library/postgres:15
 => [db internal] load .dockerignore
 => => transferring context: 2B
 => [db 1/3] FROM docker.io/library/postgres:15@sha256:5ab68e212eab9cd4a16ecbf40d9449c88e1073abdca3ecc3aa5514d4a1af2ed0
 => => resolve docker.io/library/postgres:15@sha256:5ab68e212eab9cd4a16ecbf40d9449c88e1073abdca3ecc3aa5514d4a1af2ed0
 => [db internal] load build context
 => => transferring context: 158B
 => CACHED [db 2/3] COPY postgresql.conf /etc/postgresql/postgresql.conf
 => CACHED [db 3/3] COPY init-scripts/ /docker-entrypoint-initdb.d/
 => [db] exporting to image
 => => exporting layers
 => => exporting manifest sha256:8124cbfce9388a209d14bc84be6e8102b7fd0cdc43d46b4eee7260217190797b
 => => exporting config sha256:43ce07c720306efc1c767666ec9b65ef3b5c02346b9da5265a215d90e136e50c
 => => exporting attestation manifest sha256:e25e687964b9dc9fd0de151f58380c4e419d185db89bda71f9b423a925f8b31a
 => => exporting manifest list sha256:4f761962614d6f229a54303c22f2d6f91e85c2729180fa7d1d88dcfb1dd2643e
 => => naming to docker.io/kc/postgres:15-custom
 => => unpacking to docker.io/kc/postgres:15-custom
 => [db] resolving provenance for metadata file
 => [app internal] load .dockerignore
 => => transferring context: 2B
 => [app 1/6] FROM docker.io/library/python:3.13-bookworm@sha256:6c6b3c2deae72b980c4323738be824884c9a2e17588c93db82612f8a3072be88
 => => resolve docker.io/library/python:3.13-bookworm@sha256:6c6b3c2deae72b980c4323738be824884c9a2e17588c93db82612f8a3072be88
 => [app internal] load build context
 => => transferring context: 125B
 => CACHED [app 2/6] WORKDIR /app
 => CACHED [app 3/6] COPY requirements.txt .
 => CACHED [app 4/6] RUN pip install --no-cache-dir -r requirements.txt
 => CACHED [app 5/6] COPY app.py .
 => CACHED [app 6/6] COPY /static/ static/
 => [app] exporting to image
 => => exporting layers
 => => exporting manifest sha256:f567364851a7d16a5afed0a19fc17a98d2c27af093d378ff37f99402e02e7d8b
 => => exporting config sha256:e9aa2de5510271d1c9f7d482387e947f23e1397e039d4440e7592d9e4b2f12e7
 => => exporting attestation manifest sha256:6c37dc9ad03180bee6c433f20f8db3c151209a27af7aa17a9a62348401e57e96
 => => exporting manifest list sha256:1f54a361732db23efce9b798a3f64adc64f0be6cc15c16f3d644c6de22cbbdd8
 => => naming to docker.io/kc/python:3.13-bookworm-custom
 => => unpacking to docker.io/kc/python:3.13-bookworm-custom
 => [app] resolving provenance for metadata file
[+] Running 7/7
 ✔db                                Built
 ✔app                               Built
 ✔Network kc-visit-counter_default  Created
 ✔Volume "kc-visit-counter_pgdata"  Created
 ✔Container postgres-kc-vc          Created
 ✔Container python-kc-vc            Created
 ✔Container filebeat-kc-vc          Created
Attaching to filebeat-kc-vc, postgres-kc-vc, python-kc-vc
```

Para comprobar el funcionamiento, basta con acceder desde un navegador web a http://localhost:5050 o la dirección IP de nuestro anfitrión desde cualquier dispositivo conectado en la misma red.



En la terminal, se mantendrá la salida de los contenedores, por lo que también podremos ver las peticiones HTTP que se hagan o los checkpoints de PostgreSQL.

Para evitar esto, debemos ejecutar `docker compose up --build -d` o bien sin el parámetro *--build* en caso de que ya hayamos realizado este paso, siendo `docker compose up -d` el comando a ejecutar para levantar el compose.

Si quisiéramos consultar la salida que se nos muestra sin el parámetro *-d* debemos ejecutar el comando `docker compose logs -f`

También podemos comprobar el estado del compose y sus servicios con `docker compose ls` y `docker compose ps`, así como también inspeccionar las redes o los volúmenes creados (aunque esto no aseguraría el funcionamiento de la aplicación).

Logs de la aplicación

Formato JSON a ser posible [OPCIONAL]

PostgreSQL no incorpora soporte directo para ofrecer logs en formato JSON. Para ello, es necesario incorporar a la arquitectura un transformador o parseador de logs como puede ser Filebeat, así como supone hacer importantes cambios en el proyecto.

Llegados a este punto, uno de los cambios es crear un volumen en el servicio *db* y otro en el servicio *app* llamado logs_db y logs_app a través de nuestro *docker-compose.yml*. También sería posible volcarlos todos en un mismo volumen siendo más simple conectarlo más tarde pero, por seguridad, es mejor separarlo en volúmenes independientes de manera que no puedan verse los logs de un contenedor en otro.

Una mejor solución sería extraer el contenido de los logs de contenedores (el que aparece a la ejecución de `docker logs NOMBRECONTENEDOR`), que se alojan en */var/lib/docker/containers/*/*.log*, pero desde WSL esta solución no es posible.

Cambios para adaptar el servicio de PostgreSQL a salida JSON

Creamos un *postgresql.conf* ubicado en el directorio *postgresql* con la siguiente configuración:

postgresql.conf
```
logging_collector = on
log_destination = 'stderr,jsonlog'
log_directory = '/var/log/postgresql'
log_filename = 'postgresql.log'
log_statement = 'all'
log_timezone = 'Europe/Madrid'
listen_addresses = '*'
```

Además, hacemos cambios en el servicio *db* de nuestro *docker-compose.yml*. A raíz de este momento, construiremos una imagen PostgreSQL custom teniendo como base el tag 15 de Docker Hub.

Así pues, creamos *postgres/Dockerfile* que, además de copiar el fichero de configuración, lo cargue a través del comando postgres. Igualmente, aprovechamos para copiar los scripts de inicio de creación de esquema y usuarios de base de datos.

Dockerfile
```
FROM postgres:15

# Copiar archivo de configuración
COPY postgresql.conf /etc/postgresql/postgresql.conf

# Crear y dar permisos a carpeta logs
RUN mkdir -p /var/log/postgresql \
    && chown postgres:postgres /var/log/postgresql

# Copiar scripts de inicialización
COPY init-scripts/ /docker-entrypoint-initdb.d/

# Usar el nuevo archivo postgresql.conf
CMD ["postgres", "-c", "config_file=/etc/postgresql/postgresql.conf"]
```

En nuestro *Docker-compose.yml* realizamos los cambios para que quede de esta manera, generando una imagen *kc/postgres:15-custom*.

Además, añadir un volumen que será el que más adelante permitirá leer dichos logs a otras apps de gestión de logs.

docker-compose.yml
```
services:
  db:
    build: ./postgresql
    image: kc/postgres:15-custom
    container_name: postgres-kc-vc
    env_file: .env
    volumes:
      - pgdata:/var/lib/postgresql/data
      - logs_db:/var/log/postgresql
    ports:
      - "5432:5432
(...)
volumes:
  pgdata:
  logs_db:
```

Cambios para adaptar la aplicación Python a salida JSON
Para adaptar la aplicación Python kc-visit-counter, le hemos pedido a nuestro desarrollador de confianza ChatGPT que nos adapte el código de *app.py*.

Estos logs se guardarán en */var/log/kc-visit-counter/app.log*

Posteriormente, hemos tenido que modificar el *requirements.txt*

Llegados aquí, reestructuramos el proyecto para crear un volumen exclusivo para los logs (llamado *logdata*)

docker-compose.yml

```
(...)
  app:
    build: ./app
    image: kc/python:3.13-bookworm-custom
    container_name: python-kc-vc
    env_file: .env
    ports:
      - "5050:5000"
    volumes:
      - logs_app:/var/log/kc-visit-counter
    depends_on:
      - db
(...)
volumes:
  pgdata:
  logs_db:
  logs_app:
```

Asegurarse de que todos los componentes (aplicación y base de datos) mandan sus logs por la salida estándar y salida de error (STDOUT / STDERR).

```
egz@thinkpad:~/kc-visit-counter$ docker logs python-kc-vc
{"asctime": "2025-07-29 14:22:26,389", "levelname": "INFO", "message": "Con
{"asctime": "2025-07-29 14:22:26,394", "levelname": "WARNING", "message": "
t(**DB_CONFIG)\n  File \"/usr/local/lib/python3.13/site-packages/psycopg2/_
t \"db\" (172.19.0.2), port 5432 failed: Connection refused\n\tIs the serve
{"asctime": "2025-07-29 14:22:28,423", "levelname": "INFO", "message": "\u2
 * Serving Flask app 'app'
 * Debug mode: off
{"asctime": "2025-07-29 14:22:28,433", "levelname": "INFO", "message": "\u0
l addresses (0.0.0.0)\n * Running on http://127.0.0.1:5000\n * Running on h
{"asctime": "2025-07-29 14:22:28,434", "levelname": "INFO", "message": "\u0
egz@thinkpad:~/kc-visit-counter$ docker logs postgres-kc-vc
The files belonging to this database system will be owned by user "postgres
This user must also own the server process.

The database cluster will be initialized with locale "en_US.utf8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /var/lib/postgresql/data ... ok
creating subdirectories ... ok
```

Configurabilidad de la aplicación

La aplicación debe de poder ser configurable, por ejemplo, host de la base de datos, puerto, usuario, contraseña, etc. **Mediante fichero de configuración y/o variables de entorno**. Trata de dotar de la mayor flexibilidad posible al microservicio y documenta todas las opciones de configuración en el README.

Las variables de entorno se han configurado en un fichero *.env*. Estas son las relativas a la creación del usuario, base de datos, así como la conexión a la base de datos y la zona horaria.

Para la creación del usuario de conexión a la base de datos, ha sido necesario generar un script en *postgresql/init-*

scripts/02-users.sh que obtiene las variables del contenedor, crea el usuario y le da permisos.

```
egz@thinkpad:~/kc-visit-counter$ cat postgresql/init-scripts/02-users.sh
#!/bin/bash
set -e

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" <<-EOSQL
    CREATE USER "$DB_USER" WITH ENCRYPTED PASSWORD '$DB_PASSWORD';
    GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO "$DB_USER";
    GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO "$DB_USER";
EOSQL
egz@thinkpad:~/kc-visit-counter$
```

De esta manera, no sería necesario realizar cambios en el fichero *.env* y en el fichero de creación del usuario, si no únicamente en el *.env*, simplificando la gestión y administración.

Extras

Añadir algo de logging o monitoring (para que se vean los logs centralizados, métricas con Prometheus o similar)

En la solución vamos a implementar Filebeat para parsear y enviar los logs al contenedor de Elasticsearch, que será quien almacene estos datos. Se visualizarán en otro contenedor a través de Kibana.

Añadir un contenedor Filebeat a la infraestructura

El siguiente paso es añadir un contenedor de Filebeat a nuestra infraestructura. Para ello, es necesario crear un archivo de configuración de filebeat.

En nuestro caso, hemos creado el directorio *filebeat* y, dentro de él, el *filebeat.yml* con el siguiente contenido:

filebeat.yml

```
filebeat.inputs:
  - type: log
    enabled: true
    paths:
      - /mnt/logs/app/*.log
    json.keys_under_root: true
    json.add_error_key: true
    json.message_key: message

output.console:
    pretty: true
```

En el docker-compose.yml añadimos el servicio de esta manera:

docker-compose.yml

```
(...)
  filebeat:
    image: docker.elastic.co/beats/filebeat:8.13.4
    container_name: filebeat-kc-vc
    volumes:
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - ${PWD}/filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml:ro
    depends_on:
      - db
      - app

volumes:
  pgdata:
  logs:
```

En algunos sistemas operativos, es necesario dar permisos al fichero de configuración de Filebeat para que el contenedor arranque. Ejecuta `chmod 644 filebeat/filebeat.yml`

Añadir un contenedor Elasticsearch a la infraestructura

En el *docker-compose.yml*, añadimos la imagen de *elasticsearch:9.0.4*.

docker-compose.yml

```
(...)
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.13.4
    container_name: elasticsearch-kc-vc
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    volumes:
      - es_data:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"
    networks:
      - logging
(...)
volumes:
  pgdata:
  logs_db:
  logs_app:
  portainer_data:
  es_data:
```

Como vemos, en el parámetro *xpack.security.enabled* desactivamos la autenticación para simplificar el despliegue, considerando que es una POC.

Además, se añade un volumen para que, los datos que aparecen en el dashboard o en los dataviews se guarden y permanezcan una vez que paremos y volvamos a arrancar el servicio.

Añadir un contenedor Kibana a la infraestructura

En el *docker-compose.yml*, añadimos la imagen de *kibana:9.0.4*.

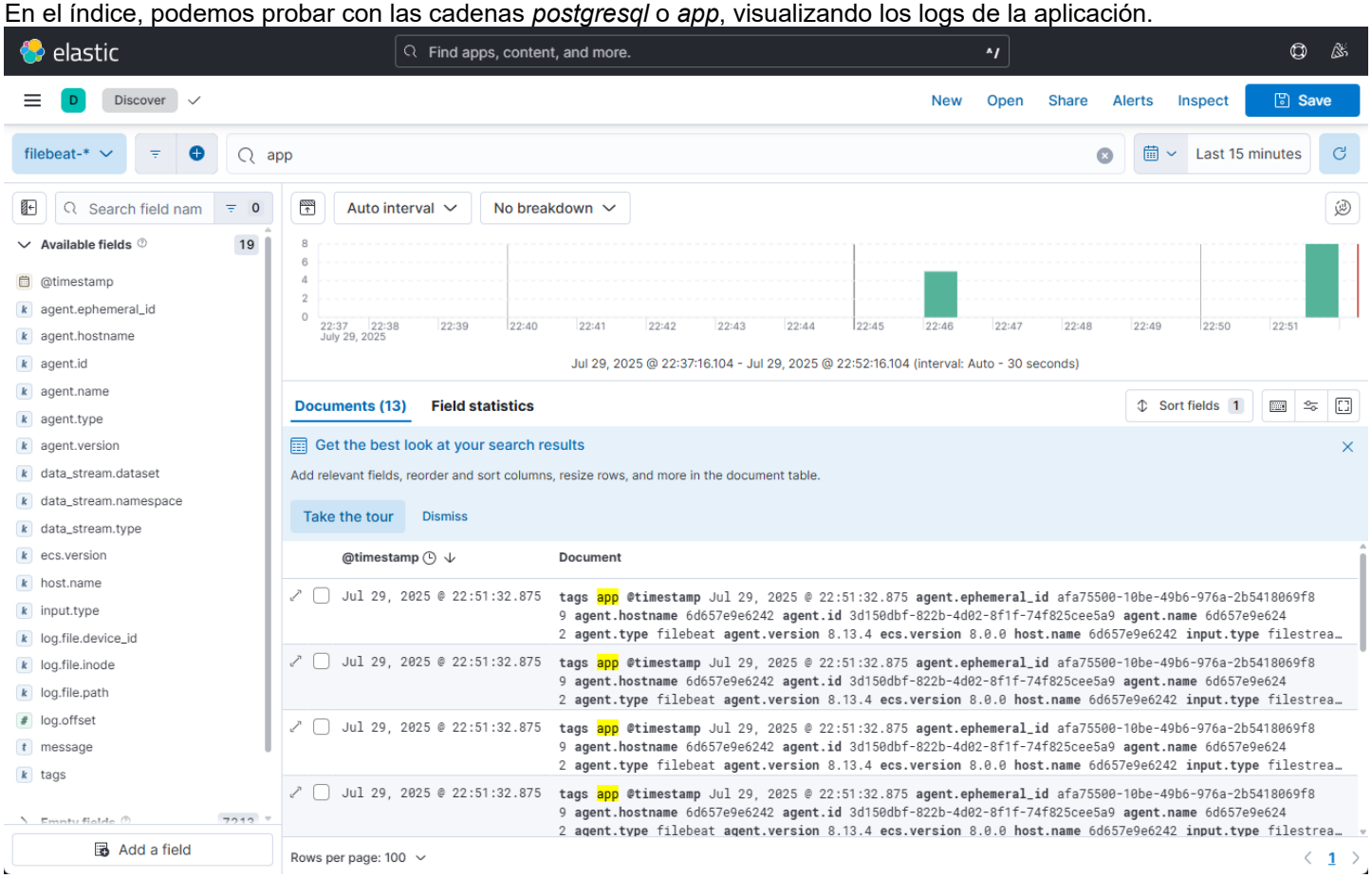
docker-compose.yml

```
(...)
  kibana:
    image: docker.elastic.co/kibana/kibana:9.0.4
    container_name: kibana-kc-vc
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch
(...)
```

Crear un "Data View" en Kibana

1. Accedemos a Kibana a través de http://localhost:5601/.
2. La primera vez que iniciamos Kibana tenemos que crear el Data View.
   Accedemos en el menú lateral a Analytics>Discover.
   - Name: filebeat-*
   - Index pattern: filebeat-*
3. A la derecha aparecerá los índices que coinciden con el patrón.
4. Pulsamos sobre el botón "Save data view to Kibana".

En el índice, podemos probar con las cadenas *postgresql* o *app*, visualizando los logs de la aplicación.



Multistage es optativo pero podéis probarlo

Esta funcionalidad nos permite crear imágenes más ligeras o livianas. Sobre todo es usado en contenedores donde hay etapas de compilación o construcción, como en nuestro caso la aplicación Python.

En el caso de nuestro Dockerfile, parte de este:

Dockerfile

```
FROM python:3.13-bookworm

WORKDIR /app

# Copiar requirements e instalar
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copiar aplicación y estáticos
COPY app.py .
COPY /static/ static/

# Correr app.py con Python
CMD ["python", "app.py"]
```

Para pasar a este Dockerfile, dividido en dos stages o etapas.

En la primera, se menciona nombra la etapa como *builder* y, en la segunda, se lanza una imagen limpia en la que se hace uso de los ficheros y dependencias creadas en la etapa *builder*.

Dockerfile

```
# Stage 1: Instalar dependencias en entorno aislado
FROM python:3.13-bookworm AS builder

WORKDIR /install

COPY requirements.txt .
RUN pip install --upgrade pip && \
    pip install --prefix=/install-deps -r requirements.txt

# Stage 2: Imagen limpia
FROM python:3.13-bookworm

WORKDIR /app

# Copiar solo las dependencias ya instaladas desde builder
COPY --from=builder /install-deps /usr/local

# Copiar tu aplicación y estáticos
COPY app.py .
COPY /static/ static/

CMD ["python", "app.py"]
```

Que la imagen esté pública y accesible desde Docker Hub

Para subir la imagen a Docker Hub, ejecutamos los siguientes pasos:

1. Nos logueamos con `docker login`
2. Etiquetamos la imagen generada por nuestro *docker-compose.yml* y la nombramos con el nombre de nuestro usuario en Docker Hub, junto con la etiqueta *latest*.
3. Realizamos un `docker push evaristogz/kc-visit-counter:latest` para subirla a Docker Hub.

```
egz@thinkpad:~/kc-visit-counter$ docker login
Authenticating with existing credentials... [Username: evaristogz]

i Info → To login with a different account, run 'docker logout' followed by 'docker login'


Login Succeeded
egz@thinkpad:~/kc-visit-counter$ docker tag kc/python:3.13-bookworm-custom evaristogz/kc-visit-counter:latest
egz@thinkpad:~/kc-visit-counter$ docker images | grep evaristogz
evaristogz/kc-visit-counter                  latest
egz@thinkpad:~/kc-visit-counter$ docker push evaristogz/kc-visit-counter:latest
The push refers to repository [docker.io/evaristogz/kc-visit-counter]
ebed137c7c18: Mounted from evaristogz/python-kc
51f8af1d7d36: Mounted from evaristogz/python-kc
2effe4ca65f9: Mounted from evaristogz/python-kc
b48daf738877: Pushed
df3a591e0441: Pushed
37f838b71c6b: Mounted from evaristogz/python-kc
e5af76e56a46: Mounted from evaristogz/python-kc
63ecad1713c0: Pushed
bdba26401be1: Pushed
d54757d897db: Pushed
873a4c802874: Mounted from evaristogz/python-kc
c2e76af9483f: Mounted from evaristogz/python-kc
latest: digest: sha256:ee86d05aec3e12db1cb9b8a0ce1e9a4db1a5fa773c73ac1944305cdbbb0c0e42 size: 856
egz@thinkpad:~/kc-visit-counter$
```

Podemos acceder a https://hub.docker.com/repositories/evaristogz para comprobar que la imagen fue subida. O bien, borrar la imagen de nuestro repositorio de imágenes local para volverla a descargar mediante *docker pull*.

Utilización de variables de entorno para las versiones de los paquetes o para los accesos a la base de datos

Se ha hecho uso de variables de entorno relativas al aprovisionamiento de la base de datos y también para la cadena de conexión de la app. Igualmente, también para el timezone que debe tener el contenedor de PostgreSQL y el de Python.

El fichero .env, aunque contiene datos de ejemplo y sin impacto real, por seguir las buenas prácticas no se ha subido al repositorio de GitHub.

Basta con crear el fichero e insertarle el siguiente contenido:

.env

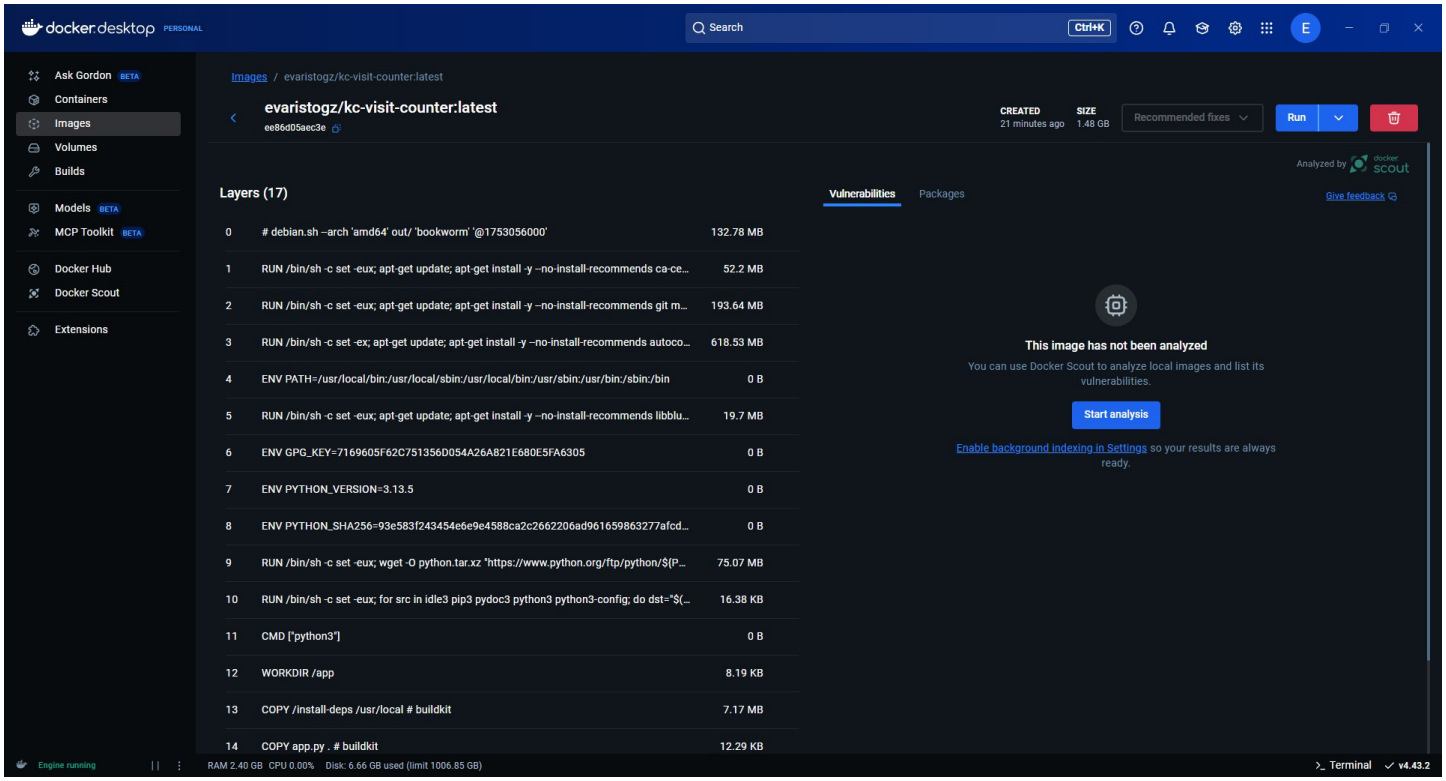
```
# Aprovisionamiento PostgreSQL
POSTGRES_USER=postgres
POSTGRES_PASSWORD=SUPER-PA$$WORD
POSTGRES_DB=kc-visit-counter

# Cadena conexión BBDD
DB_NAME=kc-visit-counter
DB_USER=kc-user
DB_PASSWORD=kc-PA$$WORD
DB_HOST=db
DB_PORT=5432

# Otros
TZ=Europe/Madrid
```

Escanear vulnerabilidades de las imágenes

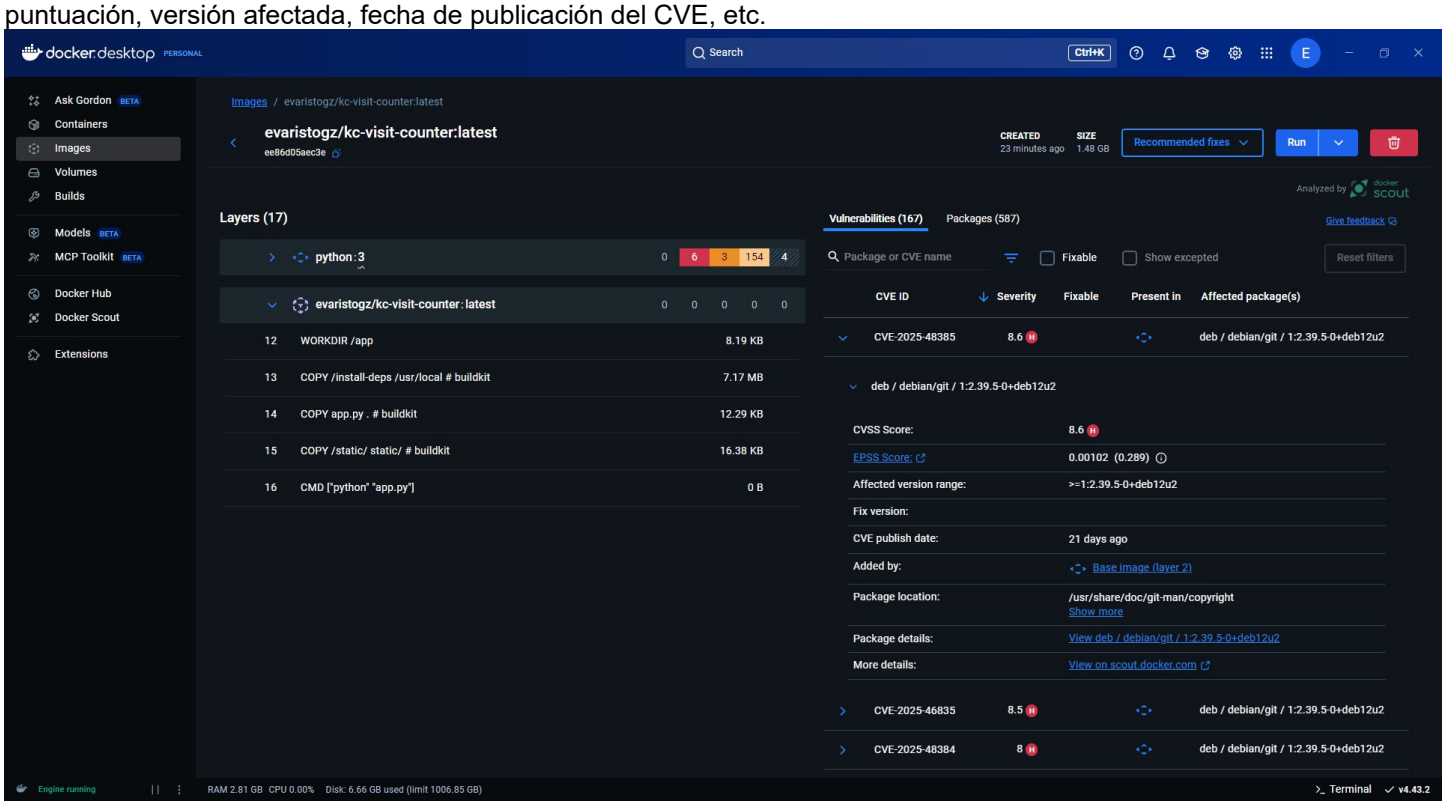
Escaneo de vulnerabilidades en imágenes mediante Docker Scout de Docker Desktop

Docker Desktop incorpora una funcionalidad de la solución Docker Scout que permite escanear las imágenes en busca de vulnerabilidades.

En este ejemplo, podemos observar las capas de la imagen que anteriormente subimos a Docker Hub.



Pulsamos sobre el botón "*Start analysis*" y, tras unos minutos, nos indicará el número de vulnerabilidades encontradas en la imagen junto a los ID de CVE o los paquetes que se ven afectados por éstas. Además de otros datos como criticidad o puntuación, versión afectada, fecha de publicación del CVE, etc.



Escaneo de vulnerabilidades en imágenes mediante la herramienta Trivy

Trivy es una herramienta de Aqua Security la cual permite escanear y buscar vulnerabilidades, configuraciones erróneas, secretos y demás en imágenes Docker. Lo hace a distintos niveles como puede ser a nivel de sistema operativo, dependencias de lenguajes de programación, paquetes, secretos o configuraciones en Dockerfile.

Existe un contenedor que permite lanzar el escáner contra la imagen que deseemos, ejecutando:
`docker run aquasec/trivy image evaristogz/kc-visit-counter:latest`

Esto genera un reporte detallado de las vulnerabilidades y su tipo.

```
egz@thinkpad:~/kc-visit-counter$ docker run aquasec/trivy image evaristogz/kc-visit-counter:latest
2025-07-29T22:43:36Z    INFO    [vulndb] Need to update DB
2025-07-29T22:43:36Z    INFO    [vulndb] Downloading vulnerability DB...
2025-07-29T22:43:36Z    INFO    [vulndb] Downloading artifact...        repo="mirror.gcr.io/aquasec/trivy-db:2"
991.15 KiB / 67.36 MiB [>_____________________________________________________] 1.44% ? p/s ?8.23 MiB / 67.36 MiB [----
->_____________________________________________] 23.27% ? p/s ?22.72 MiB / 67.36 MiB [--------------->_____________________
 44.91% 36.26 MiB p/s ETA 1s37.89 MiB / 67.36 MiB [--------------------------->_____________________] 56.25% 36.26 MiB p/s ETA
B / 67.36 MiB [------------------------------------>__________] 75.87% 36.26 MiB p/s ETA 0s58.09 MiB / 67.36 MiB [----------
-------------------------->_] 96.06% 36.09 MiB p/s ETA 0s67.36 MiB / 67.36 MiB [-----------------------------------------------------
36.09 MiB p/s ETA 0s67.36 MiB / 67.36 MiB [---------------------------------------------->] 100.00% 34.05 MiB p/s ETA 0s67.36
6 MiB [---------------------------------------------->] 100.00% 34.05 MiB p/s ETA 0s67.36 MiB / 67.36 MiB [-----------------
----------------->] 100.00% 31.85 MiB p/s ETA 0s67.36 MiB / 67.36 MiB [---------------------------------------------->] 100.00
B p/s ETA 0s67.36 MiB / 67.36 MiB [---------------------------------------------->] 100.00% 29.80 MiB p/s ETA 0s67.36 MiB / 67
----------------------------------------->] 100.00% 27.88 MiB p/s ETA 0s67.36 MiB / 67.36 MiB [--------------------------
ly downloaded   repo="mirror.gcr.io/aquasec/trivy-db:2"
2025-07-29T22:43:42Z    INFO    [vuln] Vulnerability scanning is enabled
2025-07-29T22:43:42Z    INFO    [secret] Secret scanning is enabled
2025-07-29T22:43:42Z    INFO    [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2025-07-29T22:43:42Z    INFO    [secret] Please see also https://trivy.dev/v0.64/docs/scanner/secret#recommendation for faster
2025-07-29T22:43:50Z    INFO    [python] Licenses acquired from one or more METADATA files may be subject to additional terms.
2025-07-29T22:43:58Z    INFO    Detected OS     family="debian" version="12.11"
2025-07-29T22:43:58Z    INFO    [debian] Detecting vulnerabilities...   os_version="12" pkg_num=429
2025-07-29T22:43:58Z    INFO    Number of language-specific files       num=1
2025-07-29T22:43:58Z    INFO    [python-pkg] Detecting vulnerabilities...
2025-07-29T22:43:58Z    WARN    Using severities from other vendors for some vulnerabilities. Read https://trivy.dev/v0.64/doc

Report Summary
```

Target	Type	Vulnerabilities	Secrets
evaristogz/kc-visit-counter:latest (debian 12.11)	debian	1696	-
usr/local/lib/python3.13/site-packages/MarkupSafe-3.0.2.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/blinker-1.9.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/click-8.2.1.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/flask-3.1.1.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/itsdangerous-2.2.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/jinja2-3.1.6.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/pip-25.1.1.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/psycopg2-2.9.10.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/python_json_logger-3.3.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.13/site-packages/werkzeug-3.1.3.dist-info/METADATA	python-pkg	0	-

```
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)


evaristogz/kc-visit-counter:latest (debian 12.11)
==================================================
Total: 1696 (UNKNOWN: 7, LOW: 762, MEDIUM: 653, HIGH: 263, CRITICAL: 11)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version

Desplegar Portainer o similar para ver los contenedores de forma visual (en los servidores no suele existir Docker Desktop).

Para desplegar Portainer tan solo debemos añadir un nuevo servicio a nuestro *docker-compose.yml*.

Éste vincula el socket de Docker como volumen y también crea un volumen persistente para los datos de Portainer, como puede ser el usuario y contraseña de acceso a la interfaz web.

docker-compose.yml

```
(...)
  portainer:
    image: portainer/portainer-ce:latest
    container_name: portainer-kc-vc
    restart: always
    ports:
      - "9000:9000"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - portainer_data:/data
(...)
volumes:
  pgdata:
  logs_db:
  logs_app:
  portainer_data:
```

En este caso, podemos establecer la contraseña de Portainer al acceder a http://localhost:9000/.

Sin embargo, si en los primeros 5 minutos tras levantar el contenedor no accedemos a la dirección URL y configuramos el usuario admin, Portainer deshabilita la función y obliga a reiniciar el servicio para evitar que cualquier otra persona ajena a nosotros tenga la posibilidad de establecer un usuario y contraseña y controle nuestra infraestructura Docker.

Esto únicamente puede solucionarse exponiendo los datos en el docker-compose.yml, puesto que portainer no lee automáticamente las variables del *.env*

Crear entorno de prueba y luego de producción con diferentes versiones

En este punto, vamos a establecer un entorno de prueba en un Docker Compose aparte llamado *docker-compose_test.yml* en el que se incorpore una nueva funcionalidad como puede ser tres nuevas redes para la comunicación interna de los contenedores, aumentando así la seguridad.

El entorno de producción, definido en *docker-compose.yml*, permanecerá sin redes.

En la siguiente definición podemos ver que se creará las redes *frontend*, *backend* y *monitoring* que englobaría a futuro las capas de recolección de logs, monitoring, alertas, etc.

La especificación del fichero del Compose lo hacemos con *-f* ejecutando:
`docker compose -f docker-compose_test.yml up --build -d`

docker-compose_test.yml

```
services:
  db:
    build: ./postgresql
    image: kc/postgres:15-custom
    container_name: postgres-kc-vc
    env_file: .env
    volumes:
      - pgdata:/var/lib/postgresql/data
      - logs_db:/var/log/postgresql
    ports:
      - "5432:5432"
    networks:
      - backend
```

```
  app:
    build: ./app
    image: kc/python:3.13-bookworm-custom
    container_name: python-kc-vc
    env_file: .env
    ports:
      - "5050:5000"
    volumes:
      - logs_app:/var/log/kc-visit-counter
    depends_on:
      - db
    networks:
      - backend

  filebeat:
    image: docker.elastic.co/beats/filebeat:8.13.4
    container_name: filebeat-kc-vc
    volumes:
      - ${PWD}/filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml:ro
      - logs_db:/mnt/logs/postgresql:ro
      - logs_app:/mnt/logs/kc-visit-counter:ro
    depends_on:
      - db
    networks:
      - backend
      - monitoring

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.13.4
    container_name: elasticsearch-kc-vc
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
    networks:
      - monitoring

  kibana:
    image: docker.elastic.co/kibana/kibana:8.13.4
    container_name: kibana-kc-vc
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch
    networks:
      - monitoring
      - frontend

  portainer:
    image: portainer/portainer-ce:latest
    container_name: portainer-kc-vc
    env_file: .env
    restart: always
    ports:
      - "9000:9000"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - portainer_data:/data
    networks:
      - frontend

volumes:
```

```
  pgdata:
  logs_db:
  logs_app:
  portainer_data:

networks:
  frontend:
  backend:
  monitoring:
```

Añadir algo como front más bonito, puede ser un Nginx o un Apache por delante pero que tenga un mensaje más vistoso que solo la API

La aplicación Python desarrollada no tiene función de API, por lo tanto, la incorporación de este Nginx aportaría a la infraestructura:

- Una capa extra de seguridad: debido a que el puerto 5050 no estará expuesto de manera directa.
- Una mejor administración: sería más fácil agrupar varios servicios con este proxy inverso de por medio, más cómodo y fácil de administrar certificados SSL, etc.

En este caso en concreto, también será más fácil para los usuarios al acceder directamente a un puerto común como lo es el 80.

Para hacer este cambio, quitamos el port binding del servicio *app* y exponemos el puerto de la aplicación Python (5000). También incorporamos entre el servicio *app* y el servicio *filebeat* de nuestro docker-compose.yml un servicio llamado *nginx* quedando de la siguiente forma:

docker-compose_test.yml

```
(...)
  app:
(...)
    expose:
      - "5000"

  nginx:
    image: nginx:latest
    container_name: nginx-kc-vc
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - app
(...)
```

Observa que, la vinculación de puertos o port binding ha de hacerse del 80 al 80, ya que es internamente en el siguiente fichero donde se hará el proxy_pass al puerto 5000.

También es necesario pasarle el fichero .conf para Nginx como se muestra en volumes, el cual tiene el siguiente contenido:

nginx.conf

```
events {}

http {
  server {
    listen 80;

    location / {
      proxy_pass http://app:5000;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Proto $scheme;
    }
```

```
    }
}
```

Chequear el tamaño de las imágenes y ver si se puede reducir

En primer lugar, la imagen usada tiene como base Debian Bookworm, con la finalidad de que algunos de los puntos anteriores fuesen más completos como el análisis de vulnerabilidades y seguridad, o la revisión de capas y reducción del tamaño de imagen base.

Una buena práctica es usar imágenes livianas desde el principio, construyendo sobre ella únicamente lo necesario para nuestra aplicación o proyecto. Estas pueden ser imágenes slim o basadas, por ejemplo, en alpine.

Uso de la herramienta Dive para analizar tamaños de imágenes

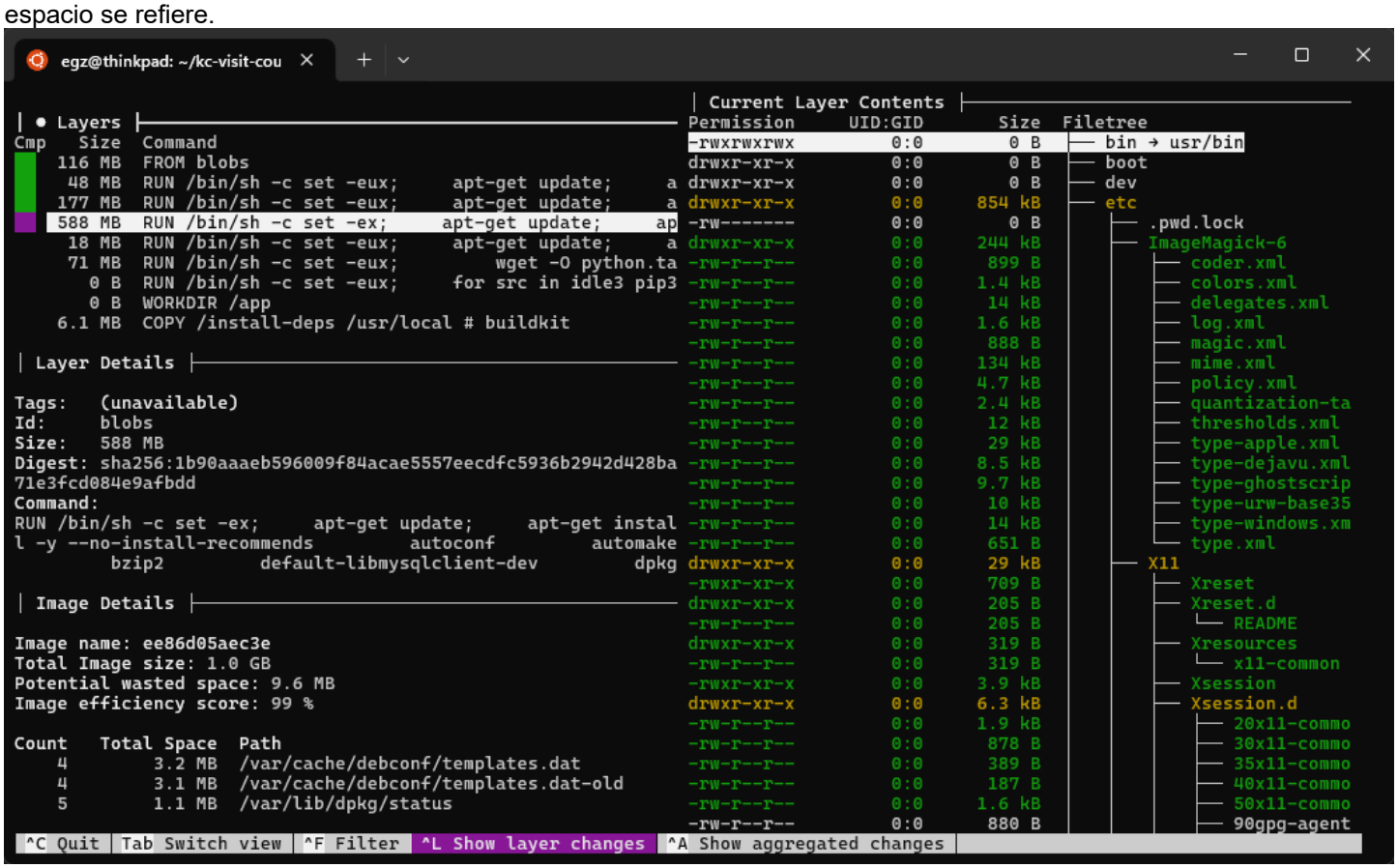
Dive es una herramienta para analizar y examinar las capas de una imagen Docker.

Su repositorio oficial es https://github.com/wagoodman/dive y su instalación está disponible para múltiples sistemas operativos.

Tras la instalación a través de su paquete *.deb*, ejecutamos `dive IdDeLaImagen`

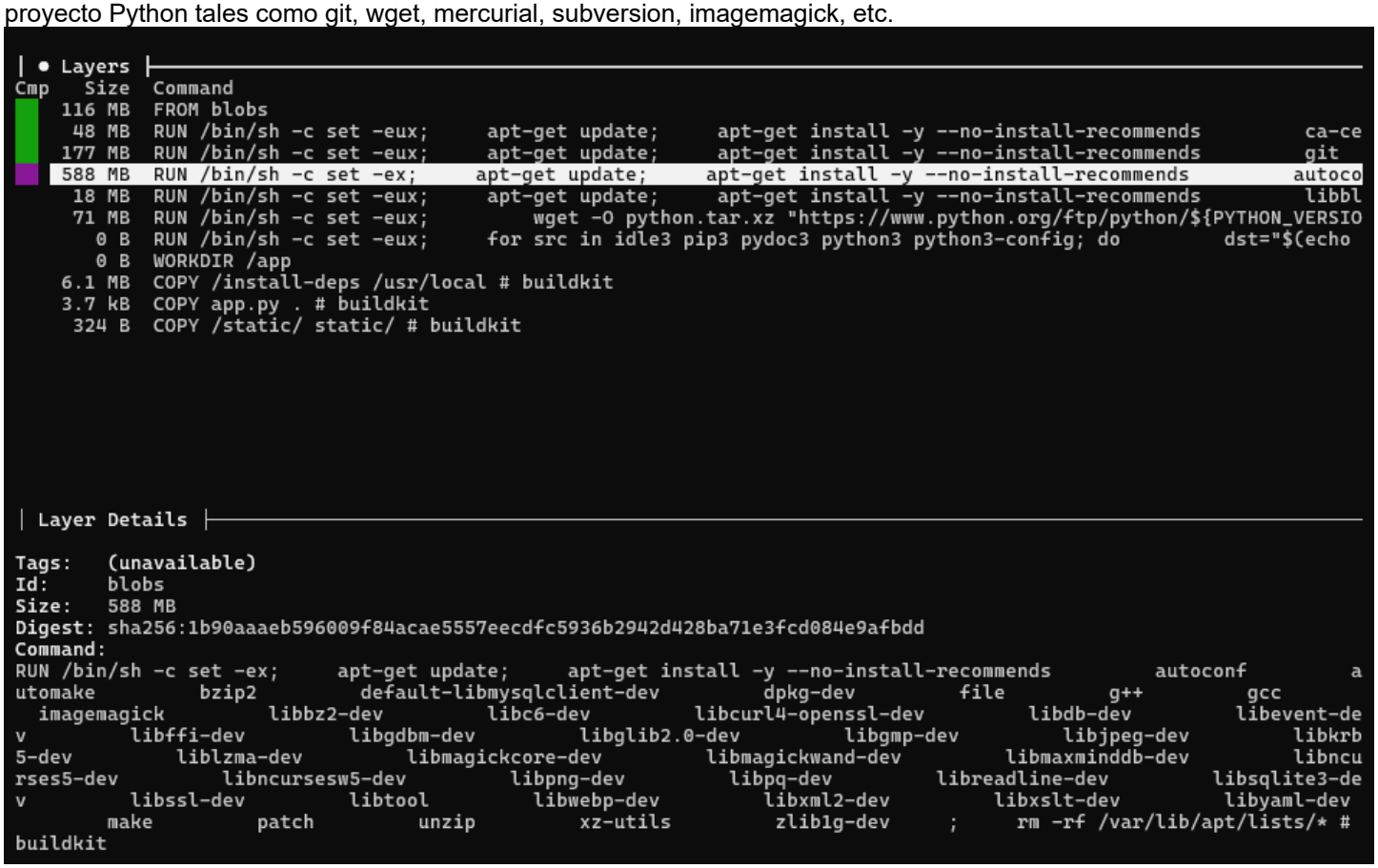
Podemos ver las distintas capas que tiene la imagen, así como el peso de cada una de ellas, su contenido, el comando ejecutado, etc.

Conforme a los detalles de la imagen, detalla que tiene un tamaño total de 1GB y una eficiencia del 99% en cuanto a espacio se refiere.



En el caso de esta que analizamos es la *evaristogz/kc-visit-counter:latest*, que es el resultado de hacer un multistage con la imagen *python:3.13-bookworm*.

Podemos observar que instala multitud de librerías, herramientas y paquetes que no nos son necesarios para nuestro proyecto Python tales como git, wget, mercurial, subversion, imagemagick, etc.



Una vez la imagen está construida, no es posible eliminar capas. Ésto solo podríamos hacerlo viendo las capas y recreando las necesarias en un Dockerfile nuevo. O bien, intentar obtener el Dockerfile base de la imagen para eliminar aquellas capas innecesarias para nuestro proyecto.

En el caso de la imagen *python:3.13-bookworm*, podemos encontrar su Dockerfile aquí: https://github.com/docker-library/python/blob/3fae0a14ac171f46e47d7ce41567e40524af5bcc/3.13/bookworm/Dockerfile.

En él, observamos que hace un FROM de *buildpack-deps:bookworm*, por lo que debemos buscar el Dockerfile de esa imagen, que se encuentra en https://github.com/docker-library/buildpack-deps/blob/d0ecd4b7313e9bc6b00d9a4fe62ad5787bc197ae/debian/bookworm/Dockerfile donde también podemos analizar que instala un total de 40 paquetes a través de *apt* y a su vez hace un FROM de *buildpack-deps:bookworm-scm*.

El Dockerfile de esa imagen está en https://github.com/docker-library/buildpack-deps/blob/d0ecd4b7313e9bc6b00d9a4fe62ad5787bc197ae/debian/bookworm/scm/Dockerfile y también hace otro FROM, en este caso de *buildpack-deps:bookworm-curl*.

En el Dockerfile *buildpack-deps:bookworm-curl* es donde encontramos, definitivamente, que se usa la imagen oficial de Debian *debian:bookworm* para la construcción: https://github.com/debuerreotype/docker-debian-artifacts/blob/c6274d8b402b7e394a69b9e0496df4c0df5efffe/bookworm/Dockerfile

Reestructurar el Dockerfile de app desde una imagen Alpine Linux

Llegados a este punto, lo más fácil, rápido, cómodo y óptimo, será cambiar nuestra imagen de Python por una de Alpine Linux. Por lo tanto, en el Dockerfile alojado en el directorio *app* de nuestro repositorio, realizamos los cambios quedando así:

Dockerfile

```
# Stage 1: Instalar dependencias en entorno aislado
FROM python:3.13-alpine AS builder
```

```
WORKDIR /install

# Instalar paquetes para compilar
RUN apk add --no-cache gcc musl-dev postgresql-dev

COPY requirements.txt .
RUN pip install --upgrade pip && \
    pip install --prefix=/install-deps -r requirements.txt

# Stage 2: Imagen limpia
FROM python:3.13-alpine

WORKDIR /app

# Instalar biblioteca PostgreSQL
RUN apk add --no-cache libpq

# Copiar solo las dependencias ya instaladas desde builder
COPY --from=builder /install-deps /usr/local

# Copiar tu aplicación y estáticos
COPY app.py .
COPY /static/ static/

CMD ["python", "app.py"]
```

También es importante cambiar el tag que se le asigna a la imagen a través del *docker-compose.yml*, en concreto en el servicio *app* reemplazando *image: kc/python:3.13-bookworm-custom* por *image: kc/python:3.13-alpine-custom*

El resultado podemos verlo aquí de manera simple ejecutando `docker images`, donde vemos:

- *kc/python*: imagen final generada a través de multistage con python:3.13-alpine como base y etiquetada como kc/python:3.13-alpine-custom desde el docker-compose.yml
- *evaristogz/kc-visit-counter*: imagen generada a través de multistage con *python:3.13-bookworm* como base y subida a GitHub.
- *python:3.13-bookworm*: imagen original y oficial con la que comenzamos este proyecto.

```
egz@thinkpad:~/kc-visit-counter$ docker images
REPOSITORY                                TAG                IMAGE ID       CREATED          SIZE
kc/python                                 3.13-alpine-custom 2210e8a267e4   12 minutes ago   79.2MB
evaristogz/kc-visit-counter               latest             ee86d05aec3e   2 days ago       1.48GB
kc/postgres                               15-custom          [illegible]    [illegible]      [illegible]
nginx                                     latest             [illegible]    [illegible]      [illegible]
portainer/portainer-ce                    latest             [illegible]    [illegible]      [illegible]
python                                    3.13-bookworm      6c6b3c2deae7   7 weeks ago      1.47GB
docker.elastic.co/kibana/kibana           8.13.4             [illegible]    [illegible]      [illegible]
docker.elastic.co/elasticsearch/elasticsearch 8.13.4         [illegible]    [illegible]      [illegible]
docker.elastic.co/beats/filebeat          8.13.4             [illegible]    [illegible]      [illegible]
egz@thinkpad:~/kc-visit-counter$
```

De esta manera, se ha logrado pasar de una imagen de 1.5GB a una imagen de 80MB.