

# Introducción a la clase

## Comenzamos

Esta clase te introduce a las aplicaciones basadas en contenedores en Docker, enseñándote los conceptos fundamentales y su importancia en la ingeniería de datos para resolver desafíos de compatibilidad al llevar software a producción. Aprenderás a construir, ejecutar y gestionar imágenes y contenedores con Docker, así como a compararlos con arquitecturas serverless e integrarlos con servicios en la nube como Amazon ECR, AWS App Runner y AWS Lambda

¡Vamos allá! 

## Objetivos de aprendizaje

- Comprender los conceptos fundamentales de contenedores y su importancia en la ingeniería de datos.
- Aprender a construir, ejecutar y gestionar imágenes y contenedores con Docker.
- Explorar el uso de redes, volúmenes y Docker Compose para definir entornos reproducibles.
- Comparar enfoques entre contenedores y serverless, e integrar contenedores con servicios cloud como ECR.

## Caso de uso integrador

¿Recuerdas el log analyzer que desarrollamos en clases pasadas?

👉 [https://docs.google.com/document/d/1MCh4Bt0VXOZMTkZGUspH8SEoWOnSviCbQBuywe9\\_nXo/edit?tab=t.klplqf70bmuu#heading=h.2fxdyijay87i](https://docs.google.com/document/d/1MCh4Bt0VXOZMTkZGUspH8SEoWOnSviCbQBuywe9_nXo/edit?tab=t.klplqf70bmuu#heading=h.2fxdyijay87i)

Pues bien, tu jefe quedó encantado con el funcionamiento mostrado, ahora necesita que lo disponibilizes para que todos los logs de los sensores de Umbrella Corp puedan ser analizados por el sistema.

Para ello tu equipo te ha sugerido usar una tecnología la cual han escuchado es altamente usada en el mercado para este tipo de casos: Docker, veamos de qué va!

# Aplicaciones basadas en contenedores en Docker

## Fundamentos de Contenedores y Docker

### ¿Por qué necesitamos contenedores?

El proceso de desarrollo de software, para casi todas las disciplinas (por no decir todas) se ve de la siguiente forma:

Tenemos una aplicación, código o funcionalidad que hemos desarrollado en nuestro entorno local, nuestro computador, y ahora necesitamos llevarla a producción; en otras palabras, que dicha aplicación sea accesible por otros usuarios.

Generalmente un entorno productivo no es más que un servidor, un computador diseñado para no apagarse y que tiene más capacidad de procesamiento y memoria que un computador personal, pero si, no deja de ser un computador. Por esto es que el problema se transforma en algo "simple", llevar nuestra aplicación o desarrollo a otro computador más. ¿Fácil no?

Pues no, resulta que es bastante complejo llevar una aplicación de un computador a otro por una serie de problemas:

- La versión entre lenguajes de programación debe de ser la misma entre el entorno local y el entorno productivo.
- Algunas operaciones del sistema cambian entre sistemas operativos, esto puede ocasionar comportamientos inesperados si el entorno local no comparte sistema operativo con el entorno productivo; cabe recalcar que este problema también se presenta a nivel de versiones del mismo sistema operativo, puede que Windows 8 maneje de forma distinta ciertas operaciones a comparacion de Windows 10. Este problema puede ocasionar que el mismo código se comporte de forma diferente para Windows 10 que para Windows 8, resultando en bugs y comportamientos inesperados.
- Las librerías usadas para desarrollar la aplicación, además de tener que ser compatibles con una versión del lenguaje de programación, también son compatibles con ciertos sistemas operativos y sus respectivas versiones.

Así pues, estamos hablando del siguiente problema ¿Cómo logró encapsular el sistema operativo, el código y sus dependencias para poder llevar mi aplicación de un computador a otro y que funcione según lo esperado? Aquí es donde entran los contenedores para ayudarnos.

## ¿Qué es un contenedor?

De este modo un contenedor es un espacio lógico donde se guardan algunas herramientas del SO (sistema operativo), mas no el SO completo, la aplicación y sus respectivas dependencias; en otras palabras, te puedes imaginar un contenedor como un tupperware en donde en una sección se almacenan algunas herramientas del SO, en otra la aplicación y en otra las dependencias (librerías, datos u otras) asociadas a la aplicación. Finalmente un contenedor contiene lo mínimo necesario para que tu aplicación funcione según lo esperado en otro computador, acabando con los problemas del tipo “pero... funcionaba en mi maquina 🤦‍♂️🤦‍♂️”.



## Contenedores vs. máquinas virtuales

Te podrás estar preguntando ¿Cuál es la diferencia entre contenedores y máquinas virtuales? Dejame decirte que es una diferencia bastante marcada.

Mientras que una **máquina virtual (VM)** emula completamente un sistema operativo con su propio kernel, ocupando decenas de GB de espacio y recursos, un contenedor comparte sistema operativo con el sistema host (quien se encuentra ejecutando al contenedor) y sólo empaqueta lo necesario para ejecutar una app específica, lo que lo hace:

- Más ligero
- Más rápido de iniciar
- Más fácil de distribuir

## **Beneficios de los contenedores en la ingeniería de datos**

En la ingeniería de datos trabajamos con cantidades absurdas de datos, cantidades tan grandes que no son procesables en un entorno local, requieren de mucho más cómputo y memoria de las que tiene un computador personal. Por ello es que probamos aplicaciones y nuevas funcionalidades con una fracción pequeña de los datos (1, 5 o 10%), y luego llevamos dichas aplicaciones a un entorno productivo, un servidor externo, ya sea en la nube u on premise.

Al llevar nuestra aplicación a un entorno productivo, los contenedores nos ayudan a asegurar la reproducibilidad y portabilidad de la misma, garantizando que el programa pueda ser ejecutado en diferentes SO y que los resultados sean consistentes.

## **Arquitectura de Docker: CLI, Daemon, Docker Hub**

Hemos entendido que son y por qué son necesarios los contenedores, ahora entendamos cómo funciona la tecnología de contenedores más usada en la actualidad: Docker, por qué si, existen otras.

Veamos cuáles y cómo funcionan los principales componentes de la arquitectura de Docker.

### **Docker Engine**

El Docker Engine es el corazón de la plataforma ya que contiene el *Docker daemon* y el *Docker client*.

### **Docker daemon**

Corre o se ejecuta en la máquina host (desde donde se ejecuta Docker) y es el responsable de la gestión de todos los "objetos Docker", entre ellos se encuentran: imágenes, contenedores, redes y volúmenes.

### **Imágenes**

Las imágenes son plantillas de solo lectura que contienen toda la información de la aplicación: código, dependencias, herramientas del SO y entornos de ejecución.

### **Contenedores**

Son lo que resulta de ejecutar la imagen, la aplicación ejecutándose. Encapsulan la aplicación, sus respectivas dependencias y partes del SO, garantizando un entorno de ejecución aislado.

### **Redes**

Las redes son la forma en cómo podemos poner a "hablar" o conectarse a distintos contenedores. Te lo puedes imaginar como un túnel que comunica un contenedor con otro y en el medio pasan mensajes.

### **Volúmenes**

Dado que los contenedores son efímeros por naturaleza, no conservan la información generada durante su ejecución. En otras palabras, cada vez que un contenedor se detiene o elimina, se pierde toda la información que contenía a menos que haya sido almacenada externamente.

Por esta razón, es fundamental contar con un mecanismo que permita persistir los datos fuera del contenedor.

Ese mecanismo son los volúmenes. Los volúmenes son la solución que proporciona Docker para almacenar datos de forma persistente, independientemente del ciclo de vida del contenedor. De este modo, la información generada durante la ejecución se mantiene disponible, incluso si el contenedor se reinicia o se elimina.

### Docker client

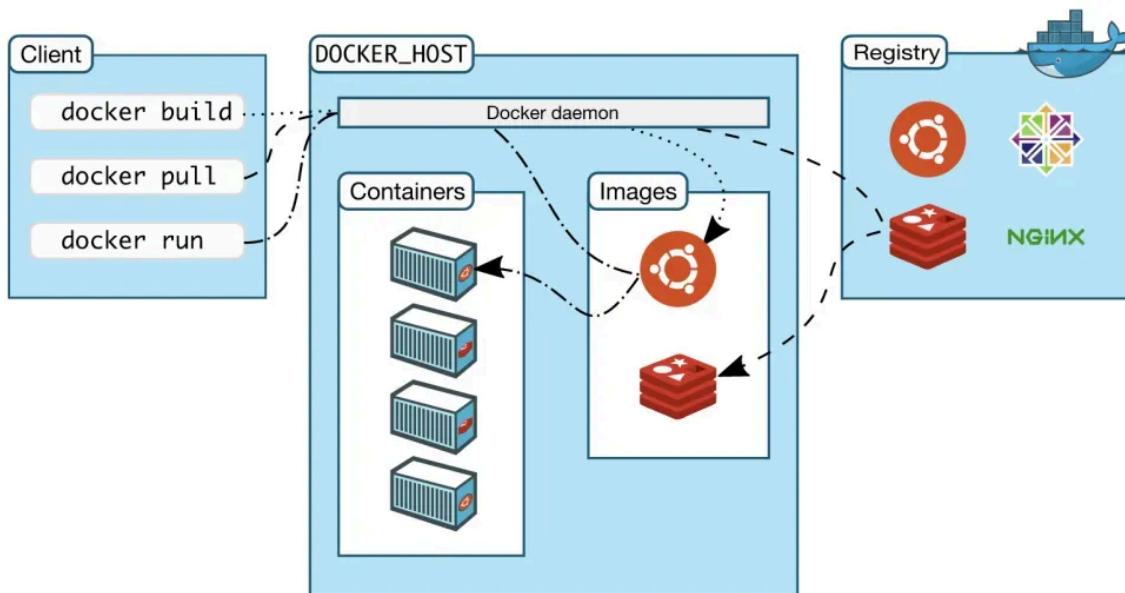
Interfaz de línea de comandos (CLI), la cual permite a los usuarios interactuar mediante comandos con el Docker daemon. Los usuarios pueden construir, ejecutar, parar o borrar "objetos Docker" por medio de la terminal.

### Docker Registry

Es un repositorio centralizado para compartir **imágenes** de Docker, en otras palabras, el GitHub de Docker. Por defecto este registry es **Docker Hub**, sin embargo, existen otras formas de compartir imágenes Docker.

Pensá en el Docker engine como una cocina:

- El *CLI* es el chef que da instrucciones.
- El *Daemon* es la cocina y el equipo que ejecuta esas órdenes.
- *Docker Hub* es como un supermercado donde conseguimos recetas ya listas (imágenes) o publicamos las nuestras.



### Ciclo de vida de los contenedores Docker

Ahora que sabemos que son los contenedores, sus beneficios, y la arquitectura usada por Docker (la solución por defecto usada en la industria) veamos el ciclo de vida de los contenedores y sus respectivas fases: construcción, inicialización, ejecución, escalado y eliminación.

## Construcción

La construcción de contenedores comienza con la creación de un archivo llamado Dockerfile. En este archivo se definen las instrucciones necesarias para construir la imagen del contenedor, incluyendo configuraciones como:

- El sistema operativo base,
- La versión del lenguaje de programación a utilizar,
- El código de la aplicación, y
- Las dependencias requeridas para su ejecución.

Este archivo actúa como una receta que indica cómo debe ser construida la imagen que luego se utilizará para crear contenedores reproducibles y portables.

### Inicio\_Video1

Analicemos un ejemplo de Dockerfile

```
FROM python:3.11-slim # SO (el SO de esta imagen base es Linux) y
versión de python a usar

WORKDIR /app # definición de directorio ppal, en donde se va a copiar
el código de la aplicación

RUN pip install uv # instalamos uv, un manejador de paquetes
COPY pyproject.toml . # copiamos las librerías requeridas por nuestra
aplicación
RUN uv sync --group prod # instalamos las librerías

ENV PATH="/app/.venv/bin:$PATH"

COPY models/ ./models/ # copiamos una carpeta de datos dentro del
contenedor

COPY src/ ./src/ # copiamos nuestro código (todo el código se encuentra
dentro de la carpeta src/) dentro del contenedor
COPY fastapi_app.py . # copiamos el script principal de nuestra
aplicación (en lugar de llamarse main se llama fastapi_app.py)

ENV PORT=8080
```

```
EXPOSE 8080 # exponemos el puerto 8080 de nuestra aplicación  
CMD ["python", "fastapi_app.py"] # ejecutamos nuestra aplicación dentro del contenedor
```

## Inicialización

Luego de crear y redactar el Dockerfile tenemos que llevar dichas configuraciones a Docker, hasta ahora lo único que tenemos es un archivo que nos dice cómo queremos que sea el contenedor, ahora es tiempo de conectar dichas especificaciones con Docker, para ello hacemos uso del siguiente comando:

```
docker build -t nombre-de-la-imagen:tag .
```

Para hacer uso de este comando ten en cuenta los siguientes puntos:

1. Tienes que tener instalado Docker en tu sistema, para verificar que se encuentra instalado puedes ejecutar el siguiente comando:

```
docker run hello-world
```

2. Dentro de la terminal donde vayas a ejecutar el comando debes de encontrarte en el directorio en el cual está el Dockerfile.
3. El tag simboliza la versión (0.0.1, 0.0.2) de dicha imagen, la mejor práctica recomendada es hacer uso de Semantic Versioning (SemVer).

Para consultar más al respecto visita  <https://semver.org/>.

Finalmente, para consultar la lista de imágenes Docker disponibles en tu sistema puedes hacer uso del siguiente comando:

```
docker image ls
```

## Ejecución

Ahora que hemos registrado nuestra imagen dentro de Docker, el siguiente paso es ejecutar y crear nuestro primer contenedor 😊. Para hacerlo podemos ejecutar el siguiente comando:

```
docker run nombre-de-la-imagen:tag
```

*DISCUSIÓN:* Este comando es muy simple, Docker nos provee más configuraciones posibles para la ejecución de una imagen. Para consultar más sobre el tema visita: <https://dockermlops.collabnix.com/docker/cheatsheet/>

Finalmente, para visualizar una lista de los contenedores actualmente en ejecución y una lista de todos los contenedores disponibles en tu sistema puedes hacer uso de los siguientes comandos respectivamente:

```
docker ps # lista de contenedores en ejecución  
docker ps -a # lista de todos los contenedores en sistema.
```

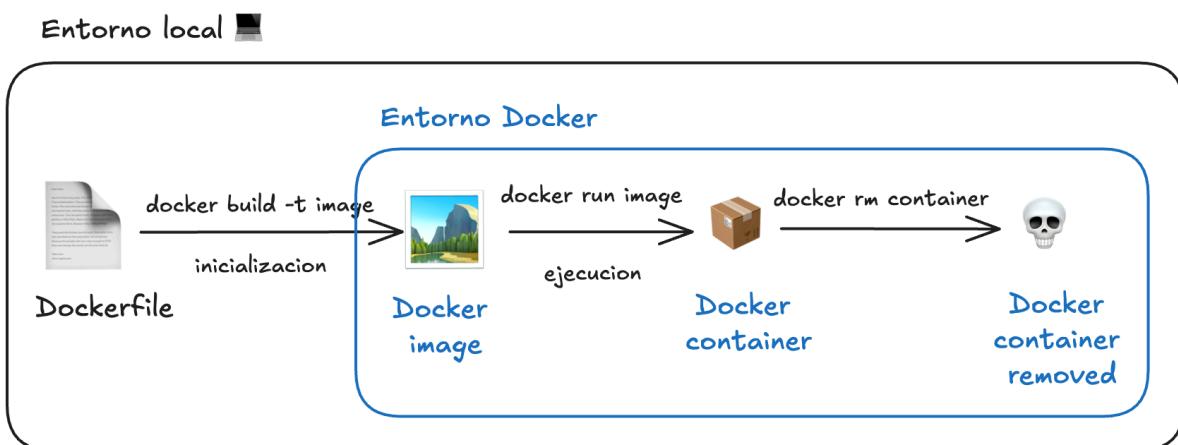
## Eliminación

La etapa final, la eliminación, consta de 2 pasos: parar la ejecución de los contenedores y posteriormente eliminarlos. Cabe recalcar que Docker nos permite eliminar también las imágenes Docker existentes.

- Para eliminar una imagen Docker existente ejecuta el siguiente comando:  
`docker rmi nombre-de-la-imagen:tag`
- El comando para poder parar la ejecución de un contenedor es el siguiente:  
`docker stop nombre-del-contenedor`
- Para eliminar un contenedor se puede hacer uso de la siguiente instrucción:  
`docker rm nombre-del-contenedor`

## Fin\_Video1

Analicemos una última vez el ciclo de vida de los contenedores Docker por medio de la siguiente figura:



1. Creamos el Dockerfile.
2. Convertimos dicho Dockerfile en una Docker image.
3. Ejecutamos la Docker image, dando como resultado la creación y ejecución de un contenedor.
4. Páramos y eliminamos el contenedor en ejecución.

## Escalado

El escalado es una etapa especial, ya que no hace parte del ciclo de vida esencial de los contenedores Docker, sin embargo, es de vital importancia hablar de ella ya que responde a qué hacer si un solo contenedor no es suficientes para satisfacer la demanda de mi aplicación.

En esencia se puede escalar de dos formas: horizontal o verticalmente.

Un escalamiento vertical implica aumentar los recursos disponibles del contenedor, aumentar la RAM, CPU y demás recursos requeridos por el contenedor para su funcionamiento; el escalado vertical tiene dos problemas, el primero de ellos siendo

que es muy caro comprar mas y mas recursos y el segundo siendo que no es posible añadir recursos ad infinitum, ya que no existen recursos infinitos.

Por otra parte, un escalamiento horizontal implica añadir más instancias o contenedores corriendo en paralelo al sistema, así ambos distribuyen la carga; este escalamiento también presenta ciertos problemas, los más prominentes siendo: como manejar la ejecución en paralelo y cómo asegurarse de que ambas instancias (contenedores) sean usadas por igual, que la carga se distribuya equitativamente entre todos los contenedores disponibles.

Lastimosamente todo este tema de escalado es un concepto de arquitectura avanzado y que se sale de los objetivos de esta clase; si quieres aprender y consultar mas sobre el tema te invito a leer el siguiente articulo:

<https://medium.com/design-microservices-architecture-with-patterns/scalability-vertical-scaling-horizontal-scaling-adb52ff679f>

## Despliegue local de nuestro analizador de logs

👉 El código requerido para esta sección se encuentra [AQUÍ](#).

Una vez que hemos repasado y comprendido nuevamente el problema previamente abordado (analizador de logs), nuestro equipo requiere que “contenericemos” – también conocido como Dockerizar– la aplicación para que así pueda ser probada y puesta en producción. ¿A que se refieren con Dockerizar? Veámoslo!

Dockerizar es el proceso por el cual tiene que pasar una aplicación para poder ser usada en forma de contenedor en Docker, anteriormente – en la sección ciclo de vida de los contenedores Docker– habíamos explicado estos pasos, sin embargo, traigamos los nuevamente a colación:

### Inicio\_Video2

#### Desarrollemos el archivo Dockerfile.

Tendremos que crear y redactar el archivo Dockerfile, el cual contiene las instrucciones y configuraciones relevantes para la creación del contenedor y funcionamiento de la aplicación.

El archivo Dockerfile para nuestra aplicación se ve de esta forma:

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt . # copiamos todas las librerías
```

```
RUN pip install --no-cache-dir -r requirements.txt # instalamos todas las  
librerías  
  
COPY data/ ./data/ # copiamos todos los datos  
COPY src/ ./src/ # copiamos todo el código fuente  
COPY main.py . # copiamos el archivo ppal  
  
ENV PORT=8000 # el puerto en el cual la app va a escuchar solicitudes  
  
EXPOSE 8000 # exponemos el puerto  
  
CMD ["python", "main.py"] # inicializamos la app
```

## Convirtamos el Dockerfile en una Docker image.

“Traslademos” las configuraciones e instrucciones de un archivo plano (Dockerfile) a Docker como tal.

Para poder crear la docker image deberás de ingresar el siguiente comando en tu terminal desde el directorio padre del proyecto:

```
docker build -t log-analyzer:0.0.1 -f docker/Dockerfile.local .
```

**IMPORTANTE:** Antes de ejecutar el comando observa bien donde debes de colocarte posicionado.

```
..ctures/DockerBasedApps + ~
~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (0.032s)
clear

~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (0.029s)
ls
data           docker          main.py        requirements.txt      src

~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (0.055s)
tree
.
├── data
│   └── logs.db
│   └── logs.txt
├── docker
│   └── Dockerfile
└── main.py
└── requirements.txt
└── src
    ├── __init__.py
    ├── application
    │   └── __init__.py
    │       └── api.py
    ├── model
    │   └── log_entry.py
    │   └── log_list.py
    └── services
        ├── __init__.py
        ├── log_pruner.py
        └── sqlite_conn.py
            └── temporal_cache.py

7 directories, 14 files

~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 A Agents ⓘ
docker build -t log-analyzer:0.0.1 -f docker/Dockerfile .
```

Te aparecerá la siguiente información en tu terminal, esta información es básicamente como se copian los archivos e instalan las librerías mencionadas en el Dockerfile.

```
~/.ctures/DockerBasedApps + ~
```

```
~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (0.055s)
tree
└── sqlite_conn.py
    └── temporal_cache.py

7 directories, 14 files
```

```
~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (1.444s)
docker build -t log-analyzer:0.0.1 -f docker/Dockerfile .

[+] Building 1.1s (13/13) FINISHED                                            docker:desktop-linux
=> [internal] load build definition from Dockerfile                         0.0s
=> => transferring dockerfile: 258B                                         0.0s
=> [internal] load metadata for docker.io/library/python:3.11-slim          1.0s
=> [auth] library/python:pull token for registry-1.docker.io                 0.0s
=> [internal] load .dockerrcignore                                         0.0s
=> => transferring context: 2B                                           0.0s
=> [1/7] FROM docker.io/library/python:3.11-slim@sha256:139020233cc412efe4c8135b0efe1c7569dc8b28ddd88bdbb109b764f89 0.0s
=> => resolve docker.io/library/python:3.11-slim@sha256:139020233cc412efe4c8135b0efe1c7569dc8b28ddd88bdbb109b764f89 0.0s
=> [internal] load build context                                         0.0s
=> => transferring context: 657B                                         0.0s
=> CACHED [2/7] WORKDIR /app                                              0.0s
=> CACHED [3/7] COPY requirements.txt .                                     0.0s
=> CACHED [4/7] RUN pip install --no-cache-dir -r requirements.txt       0.0s
=> CACHED [5/7] COPY data/ ./data/                                         0.0s
=> CACHED [6/7] COPY src/ ./src/                                         0.0s
=> CACHED [7/7] COPY main.py .                                             0.0s
=> exporting to image                                                       0.0s
=> => exporting layers                                                     0.0s
=> => exporting manifest sha256:e7afb780e930f49a2b5f61b479da2f1ec8c73b8f97fa8767e697c5baf732ec3 0.0s
=> => exporting config sha256:cc7001643cb4013fd6e61be10426675a118f289bede15ef5163ac7a9e5042c 0.0s
=> => exporting attestation manifest sha256:f3bbff435033817c24830807ad8162d6416aa3369f4f174ef4a3b4d10f59302c0 0.0s
=> => exporting manifest list sha256:0989e3af0f699a5b832bf2b9f8ac64d67cf2f7a30179c31bae90b5763cab7ed7 0.0s
=> => naming to docker.io/library/log-analyzer:0.0.1                      0.0s
=> => unpacking to docker.io/library/log-analyzer:0.0.1                     0.0s
```

```
Run the newly built docker image. ⌘ ↩
```

```
~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 ⌘ Agents ⌘ |
```

```
docker run -p 8000:8000 log-analyzer:0.0.1 ⌘
```

Listo! Al ejecutar el siguiente comando deberías de poder ver el nombre de la imagen recientemente creada – log-analyzer:0.0.1 – listada en las imágenes disponibles.

```
docker image ls
```

## Transformando la Docker image en un contenedor

Bien, ahora pongamos en marcha nuestra aplicación, ejecutando la imagen de Docker generada y transformándola en un contenedor. Para hacerlo, ejecuta el siguiente comando en tu terminal:

```
docker run -p 8000:8000 log-analyzer:0.0.1
```

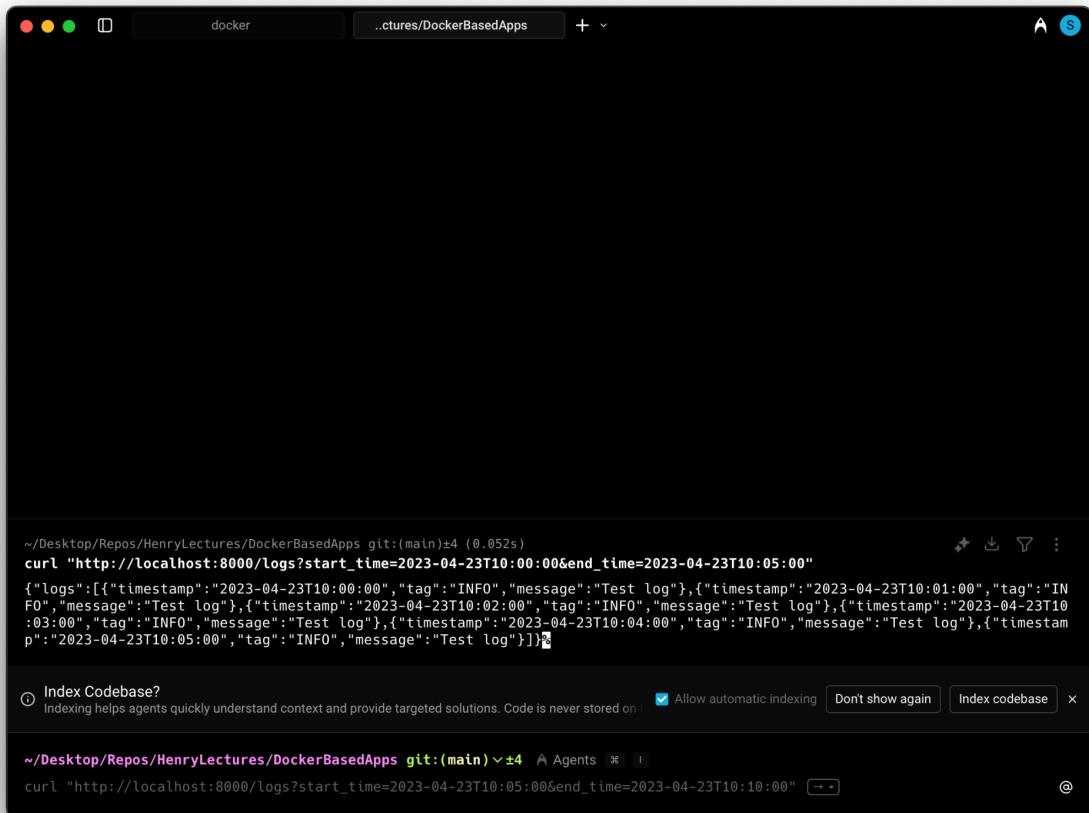
**Finalmente, probemos nuestra aplicación**

Para probar que nuestra aplicación funciona, podemos ejecutar el siguiente comando desde la terminal:

```
curl  
"http://localhost:8000/logs?start_time=2023-04-23T10:00:00&end_time=2023  
-04-23T10:05:00"
```

Este comando busca todos los logs disponibles en la aplicación dado un rango de fechas, para este caso, todos los logs existentes entre las fechas 2023-04-23 10:00:00am y 2023-04-23 10:05:00am.

Si todo funciona bien deberías de ver algo como esto en tu terminal.



```
~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (0.052s)
curl "http://localhost:8000/logs?start_time=2023-04-23T10:00:00&end_time=2023-04-23T10:05:00"
{"logs":[{"timestamp":"2023-04-23T10:00:00","tag":"INFO","message":"Test log"}, {"timestamp":"2023-04-23T10:01:00","tag":"INFO","message":"Test log"}, {"timestamp":"2023-04-23T10:02:00","tag":"INFO","message":"Test log"}, {"timestamp":"2023-04-23T10:03:00","tag":"INFO","message":"Test log"}, {"timestamp":"2023-04-23T10:04:00","tag":"INFO","message":"Test log"}, {"timestamp":"2023-04-23T10:05:00","tag":"INFO","message":"Test log"}]
```

① Index Codebase  
Indexing helps agents quickly understand context and provide targeted solutions. Code is never stored on |  Allow automatic indexing |  |  X

```
~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)✓±4 ⚡ Agents ⚡
curl "http://localhost:8000/logs?start_time=2023-04-23T10:05:00&end_time=2023-04-23T10:10:00" ↵ ↵ @
```

## Otra forma de despliegue: Docker compose

Ahora que ya conoces cómo se despliegan contenedores de forma tradicional usando `docker build` y ejecutándolos manualmente, es momento de presentarte una alternativa mucho más **elegante y escalable: Docker Compose**.

Docker Compose es una herramienta que permite **orquestar múltiples contenedores Docker** desde un único archivo de configuración. Hasta ahora, nuestra aplicación ha sido sencilla: solo necesita **un servicio** (el analizador de logs). Pero en entornos reales, las aplicaciones pueden requerir **varios servicios simultáneamente**, como bases de datos, colas de mensajes, frontends, APIs, etc.

¿Te imaginas tener que construir y ejecutar cada uno manualmente con `docker build` y `docker run`? Sería tedioso, propenso a errores y poco mantenible.

Justo para resolver este problema se ideó docker compose, la idea es que en único archivo – `docker-compose.yml` – definas todos los servicios que tu aplicación

requiere, los conectes definiendo una red y con un solo comando puedes ejecutar todos los servicios previamente definidos.

Peguemosle una mirada!

## Nuestro Docker Compose

El siguiente fragmento de código es el archivo **Docker Compose** con el cual vamos a trabajar. No te preocupes si no entiendes todo de inmediato, enseguida te explico cada parte:

```
version: '3.9'

services:
  log-analyzer:
    build:
      context: .. # root directory so it can access main.py file
      dockerfile: docker/Dockerfile.local
    ports:
      - 80:80
    environment:
      - RUNENV=local
    networks:
      - log-analyzer-network

  postgres:
    image: postgres:14-alpine
    ports:
      - 5432:5432
    volumes:
      - ~/apps/postgres:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=admin
      - POSTGRES_USER=admin
      - POSTGRES_DB=citizix_dbzzz
    networks:
      - log-analyzer-network

networks:
  log-analyzer-network:
    driver: bridge
```

Este archivo define **dos servicios**:

- **log-analyzer**, que es nuestra aplicación dockerizada.
- **postgres**, una base de datos PostgreSQL lista para usar.

Además:

- Ambos servicios están conectados a una **red común** llamada **log-analyzer-network**, lo que permite que se comuniquen entre sí (por ejemplo, que **log-analyzer** pueda conectarse a la base de datos usando el nombre **postgres** como hostname).

- El servicio `log-analyzer` construye su imagen desde un Dockerfile específico (`Dockerfile.local`) ubicado dentro de la carpeta `docker`, pero usando como contexto la raíz del proyecto (`..`) para poder acceder a `main.py` y a todo el código.
- Finalmente, se exponen los puertos correspondientes para poder interactuar con la app (puerto 80) y con la base de datos (puerto 5432) desde fuera del contenedor.

En resumen: un entorno completo y funcional, definido en un solo archivo.

## Ejecutemos nuestro Docker Compose

Ejecutar nuestro **Docker Compose** es bastante sencillo.

Primero, asegúrate de estar ubicado en la carpeta raíz del proyecto —en este caso, `DockerBasedApps`— desde tu terminal o consola.

Una vez allí, ejecuta el siguiente comando para levantar todos los servicios definidos:

```
docker compose -f docker/docker-compose.yml up -d
```

Este comando le dice a Docker que use el archivo `docker-compose.yml` que está dentro de la carpeta `docker`, y que levante los servicios en segundo plano (-d significa `detached`).

Para verificar que todo está corriendo correctamente, ejecuta:

```
docker ps
```

Con esto verás una lista de todos los contenedores activos. Deberías ver tu aplicación `log-analyzer` y el contenedor de `postgres` funcionando sin problema.

¡Listo! Acabas de levantar una arquitectura multi-servicio con un solo comando 🎉

```

~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (0.298s)
docker compose -f docker/docker-compose.yml up -d
WARN[0000] /Users/spuertaf/Desktop/Repos/HenryLectures/DockerBasedApps/docker/docker-compose.yml: the attribute 'version' is obsolete, it will be ignored, please remove it to avoid potential confusion
[+] Running 3/3
  ✓ Network docker_log-analyzer-network Created
  ✓ Container docker-log-analyzer-1 Started
  ✓ Container docker-postgres-1 Started
          0.0s
          0.1s
          0.1s

~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4 (0.075s)
docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED          STATUS          PORTS          NAMES
fffb8973cf1   postgres:14-alpine "docker-entrypoint.s..." 3 minutes ago   Up 3 minutes   0.0.0.0:5432->5432/tcp, [::]:5432->5432/tcp   docker-postgres-1
0738e6627a4d   docker-log-analyzer   "python main.py"   3 minutes ago   Up 3 minutes   0.0.0.0:80->80/tcp, [::]:80->80/tcp   docker-log-analyzer-1
f3e4d40b8b43   moby/buildkit:buildx-stable-1 "buildkitd --allow-stdin" 3 days ago    Up About an hour   buildx.buildkit_gooify_difff

  A Help me manage running containers.  ⌂
~/Desktop/Repos/HenryLectures/DockerBasedApps git:(main)±4  ⌂ Agents  ⌂
Docker logs docker-log-analyzer-1 ⌂ @

```

## Fin\_Video2

### Un siguiente reto

¡Excelente! Hemos logrado dockerizar (contenerizar) y probar exitosamente nuestra aplicación en un entorno local 🎉.

El siguiente paso es llevar esta aplicación a un entorno productivo, donde pueda ser utilizada por diversos usuarios de forma remota.

Es importante entender que, en su estado actual, la aplicación solo puede ser accedida desde tu entorno local, o por cualquier persona que ejecute manualmente la imagen de Docker en su propia máquina.

Para que usuarios externos puedan acceder a ella —por ejemplo, desde un navegador o a través de Internet— necesitaremos desplegarla en una infraestructura que permita el acceso público.

Existen principalmente dos formas de hacerlo:

1. En un servidor on-premise, es decir, un servidor físico o virtual administrado directamente por la empresa, con acceso público pre configurado.

- O montando la aplicación en un entorno en la nube ☁, lo cual ofrece mayor flexibilidad, escalabilidad y menor mantenimiento operativo.

Como nuestro equipo ya tiene gran parte de su infraestructura en la nube, lo más natural es desplegar nuestra aplicación de análisis de logs allí también.

Ahora bien, surge una última pregunta:  
¿Cómo desplegar la aplicación en la nube?

Lamentablemente, la respuesta no es tan sencilla. Existen dos enfoques principales: **Serverless** y **Contenedores**.

¡Veámoslo a continuación!

## Arquitecturas Serverless vs Contenedores

Hemos visto como Docker presenta una solución de como llevar mi aplicación desde un entorno local a uno productivo, sin embargo, existe otra solución a este problema: serverless.

### ¿Qué es serverless?

**Serverless** es un modelo de computación en el cual el usuario sube únicamente el código fuente y sus dependencias, sin preocuparse por la infraestructura subyacente. El proveedor de nube se encarga automáticamente del aprovisionamiento, escalado, despliegue y ejecución del servicio. Veamos una comparación más a fondo en la siguiente tabla.

	Serveless	Contenedores
Gestión de la infraestructura	El cloud provider la gestiona.	Gestionada por vos.
Duración de ejecución	La idea es que serverless sea usado para la ejecución de funciones finitas y bien definidas, por lo tanto, el tiempo máximo de ejecución es limitado.	Ilimitado
Flexibilidad	Limitada, puedes configurar lo que el proveedor te deje configurar para la ejecución de tu aplicación.	Ilimitada, ya que gestionas tu la infraestructura puedes configurar lo que deseas de la ejecución de tu aplicación
Casos de uso	Funciones pequeñas y	Aplicaciones complejas,

	bien definidas, API's.	servicios de almacenamiento de datos.
--	------------------------	---------------------------------------

Finalmente, cabe resaltar que serverless y contenedores no son modelos excluyentes. De hecho, muchos proveedores de nube —por no decir todos— empaquetan internamente el código y las dependencias del usuario en contenedores para su ejecución.

La diferencia fundamental radica en quién es responsable de gestionar la infraestructura y el ciclo de vida del contenedor (inicialización, ejecución, escalado, eliminación).

En el modelo serverless, esa responsabilidad recae completamente en el proveedor de nube, mientras que cuando ejecutas contenedores directamente, tú como usuario debes asumir ese control.

## Orientación a eventos

Una particularidad de un despliegue serverless es su **orientación a los eventos**, un evento es una acción que acontece en el mundo real identificada con un tiempo de ocurrencia.

Veamos los siguientes ejemplos de eventos:

```
{
  "event_type": "loza.sucia.detectada",
  "timestamp": "2025-06-30T21:00:00Z",
  "data": {
    "ubicacion": "cocina",
    "cantidad_elementos": 12,
    "usuario_responsable": "Santiago"
  }
}
```

```
{
  "event_type": "loza.lavado.iniciado",
  "timestamp": "2025-06-30T21:05:00Z",
  "data": {
    "usuario": "Santiago",
    "modo": "manual",
    "cantidad_elementos": 12
  }
}
```

```
{  
  "event_type": "loza.elemento.lavado",  
  "timestamp": "2025-06-30T21:06:45Z",  
  "data": {  
    "elemento": "plato",  
    "estado": "limpio",  
    "detergente_utilizado": true  
  }  
}
```

```
{  
  "event_type": "loza.almacenada",  
  "timestamp": "2025-06-30T21:15:00Z",  
  "data": {  
    "cantidad_elementos": 12,  
    "ubicacion": "gabinete_superior"  
  }  
}
```

De este modo, eventos de consultas a páginas web se podrían ver de la siguiente manera:

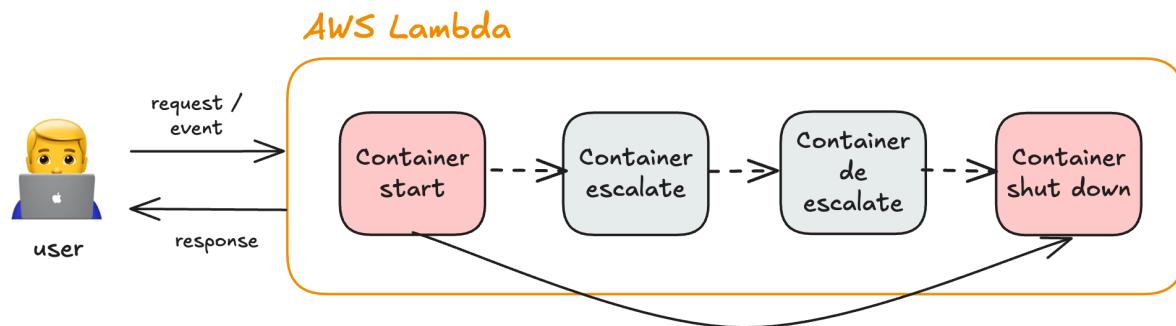
```
{  
  "event_type": "pagina.consultada",  
  "timestamp": "2025-06-30T21:30:00Z",  
  "data": {  
    "usuario_id": "user_12345",  
    "pagina_url": "https://www.ejemplo.com/productos/123",  
    "metodo_http": "GET",  
    "ip_origen": "192.168.1.25",  
    "navegador": "Chrome",  
    "dispositivo": "desktop",  
    "referer": "https://www.ejemplo.com/inicio",  
    "duracion_estimada_segundos": 15  
  }  
}
```

Cuando hablamos de orientación a eventos en arquitecturas serverless, nos referimos a que los sistemas están diseñados para “escuchar” ciertos tipos de eventos y reaccionar automáticamente cuando estos ocurren.

Por ejemplo, cuando se detecta un evento del tipo `"pagina.consultada"`, la función alojada en AWS Lambda se activa automáticamente para procesar esa solicitud. No

hace falta que la aplicación esté encendida permanentemente; solo se activa en respuesta a los eventos que le interesan.

De esta forma, el funcionamiento de nuestra aplicación de análisis de logs al ser desplegada de forma serverless se vería de la siguiente forma:



1. El usuario genera un evento al consultar (tratar de usar) la aplicación alojada en AWS lambda.

```
{  
  "event_type": "pagina.consultada",  
  "timestamp": "2025-06-30T21:30:00Z",  
  "data": {  
    "usuario_id": "user_12345",  
    "pagina_url": "https://www.ejemplo.com/productos/123",  
    "metodo_http": "GET",  
    "ip_origen": "192.168.1.25",  
    "navegador": "Chrome",  
    "dispositivo": "desktop",  
    "referer": "https://www.ejemplo.com/inicio",  
    "duracion_estimada_segundos": 15  
  }  
}
```

2. AWS Lambda escucha o detecta dicho evento e inicializa la aplicación contenerizada
3. [OPCIONAL] Si AWS lambda detecta que los recursos de la aplicación no serán suficientes para responder a las peticiones de los usuarios, escalara la aplicación de forma vertical – asignará más CPU y memoria a la aplicación –.
4. [OPCIONAL] Cuando AWS detecte que existe un exceso de recursos en la aplicación para satisfacer a las peticiones actuales, realizará lo opuesto al paso anterior, de escalara (quitará) recursos de la aplicación hasta que sean suficientes para satisfacer las peticiones de los clientes.
5. Finalmente, cuando se detecte que la necesidad puntual del usuario que realizó la petición fue satisfecha, AWS Lambda parará la ejecución del contenedor. La

instancia puede mantenerse en "espera" durante un tiempo por si hay nuevas invocaciones, pero eventualmente será eliminada.

## Integración de Contenedores con la Nube: ECR

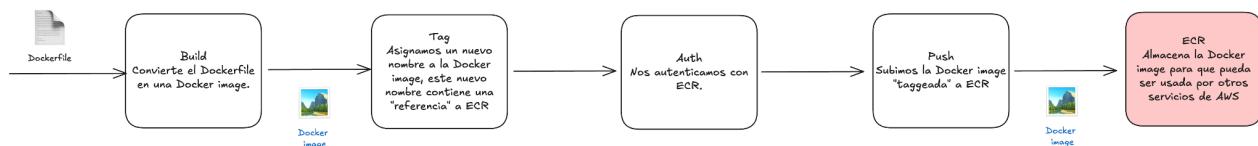
Antes de poder proseguir a desplegar nuestra aplicación en servicios de la nube, ya que esta se encuentra contenerizada, debemos de guardarla en un punto intermedio, este punto intermedio se llama Elastic Container Registry - ECR -.

### ¿Qué es ECR?

ECR es un registro de contenedores completamente administrado que ofrece alojamiento de alto rendimiento, lo que permite utilizar imágenes de aplicaciones y artefactos de forma confiable en cualquier lugar, en otras palabras, es un lugar administrado por AWS en donde almacenamos nuestras imágenes de Docker para que otros servicios de AWS puedan usarlas.

### Flujo CI/CD básico: build → tag → push → deploy

El flujo para poder llevar un Dockerfile a ECR es el siguiente:



Por ahora ten en mente los pasos y propósitos que aparecen en la figura, a continuación los veremos en acción.

## Despleguemos nuestra aplicación en la nube 😊

👉 El código requerido para esta sección se encuentra [AQUÍ](#).

Ya que conocemos ambas estrategias de despliegue – contenedores o serveless – es momento de desplegar nuestra aplicación en la nube para consumo de los usuarios. Veamos cómo desplegar y probar el funcionamiento de nuestra aplicación, finalmente determinemos cuál de las dos estrategias sería la más "correcta".

### Desplegando la aplicación en AWS App Runner

#### ¿Qué es AWS App Runner?

App runner es un servicio especializado para el despliegue de aplicaciones contenerizadas; se encarga de la **ejecución** y **escalado** de los servicios. Es importante tener en cuenta que este servicio **no se encarga** de la **eliminación** del servicio, por lo cual esta responsabilidad es delegada al desarrollador.

Sin embargo, antes de desplegar allí debemos de llevar nuestra aplicación "Dockerizada" a un servicio de AWS llamado ECR, recordemos que ECR es un servicio para almacenar imágenes de Docker, de esta forma, una vez almacenada la imagen en ECR otros servicios pueden hacer uso de ellas.

## Inicio\_Video

### Convertamos el Dockerfile en una Docker image

Transformemos el Dockerfile.lambda en una Docker image, mediante el siguiente comando:

```
docker build --provenance=false -t log-analyzer:0.0.1 -f  
docker/Dockerfile.local .
```

### Creemos un repositorio de Amazon ECR

Debemos de crear un repositorio, un lugar, en donde AWS almacenará nuestra imagen de Docker:

```
aws ecr create-repository --repository-name log-analyzer --region  
us-east-1
```

El comando te retornará algo de este estilo:

```
{  
    "repository": {  
        "repositoryArn":  
            "arn:aws:ecr:us-east-1:128624919537:repository/log-analyzer",  
            "registryId": "128624919537",  
            "repositoryName": "log-analyzer",  
            "repositoryUri":  
                "128624919537.dkr.ecr.us-east-1.amazonaws.com/log-analyzer",  
            "createdAt": "2025-07-03T21:18:32.088000-05:00",  
            "imageTagMutability": "MUTABLE",  
            "imageScanningConfiguration": {  
                "scanOnPush": false  
            },  
            "encryptionConfiguration": {  
                "encryptionType": "AES256"  
            }  
    }  
}
```

### Autenticandonos en ECR

Ahora tendremos que autenticarnos con ECR para posteriormente cargar la Docker image a este servicio.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 128624919537.dkr.ecr.us-east-1.amazonaws.com
```

**IMPORTANTE:** Recuerda reemplazar

**128624919537.dkr.ecr.us-east-1.amazonaws.com** por el valor retornado en la creación del repositorio ECR.

### Etiquetemos nuestra Docker image

Luego de creado el repositorio en Amazon ECR y habernos autenticado con dicho servicio, tendremos que asignar una etiqueta a nuestra imagen de Docker local de la siguiente forma:

```
docker tag log-analyzer:0.0.1  
128624919537.dkr.ecr.us-east-1.amazonaws.com/log-analyzer:0.0.1
```

**IMPORTANTE:** Recuerda reemplazar

**128624919537.dkr.ecr.us-east-1.amazonaws.com** por el valor retornado en la creación del repositorio ECR.

### Subamos la Docker image a Amazon ECR

Llevemos la imagen de nuestro entorno local a Amazon ECR por medio del siguiente comando:

```
docker push  
128624919537.dkr.ecr.us-east-1.amazonaws.com/log-analyzer:0.0.1
```

**IMPORTANTE:** Recuerda reemplazar

**128624919537.dkr.ecr.us-east-1.amazonaws.com** por el valor retornado en la creación del repositorio ECR.

La práctica recomendada al usar ECR es crear un repositorio ECR por cada aplicación diferente que tengamos, otra forma de verlo, es crear un repositorio diferente para cada Dockerfile distinto que manejemos.

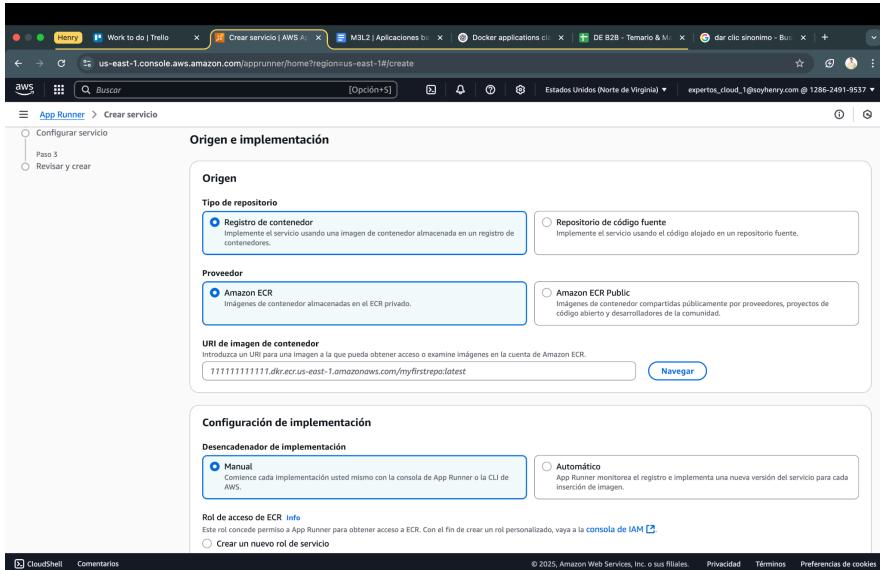
### Fin\_Video

Una vez que hayamos subido nuestra imagen de Docker a ECR, podremos desplegar nuestra aplicación; te recomendamos que para los siguientes pasos hagas uso de la plataforma web de AWS.

### Creando un nuevo servicio de App Runner

1. Una vez logueado en la interfaz web de AWS, busca por el servicio "AWS App Runner" en la barra de búsqueda.
2. Presiona sobre el servicio mencionado para abrir su panel de opciones.
3. Cliquea el botón "Crear servicio".

- Bajo el panel “Origen e implementación”, en la opción “URI de imagen de contenedor” presiona el botón “navegar”.



- Busca el repositorio creado: “log-analyzer” y selecciona la versión de la imagen subida: 0.0.1. Presiona “Continuar”.
- Bajo el panel “Configuración de implementación”, en la opción “Desencadenador de implementación” déjalo como está – manual –. Posteriormente en la opción “Rol de acceso de ECR” selecciona la opción “Crear un nuevo rol de servicio”.
- Da clic en “Siguiente”.
- Bajo el panel “Configuración del servicio” asigne un nombre al servicio a crear, para este caso le asigne el nombre: “log-analyzer-v1”.
- Bajo el panel “Configuración del servicio” cambia el valor de “Puerto” de 8080 a 80.
- Deja lo demás tal cual está y presiona sobre el botón “Siguiente”.
- Finalmente, haz clic sobre el botón “Crear e implementar”.
- Si todo salió bien, el siguiente mensaje debería de aparecer en la interfaz del servicio recién creado.

The screenshot shows the AWS App Runner service details page for a service named 'log-analyzer-v1'. The top navigation bar includes tabs for 'Servicios' and 'Dashboard'. Below the header, a message says 'Implementación en log-analyzer-v1 en curso. Esto puede tardar varios minutos.' The main section is titled 'Información general del servicio' and displays the following information:

- Estado:** Operation in progress
- ARN del servicio:** arn:aws:apprunner:us-east-1:128624919537:service/log-analyzer-v1/32345dd0c4074af9ac79f0a5122c1351
- Origen:** 128624919537.dkr.ecr.us-east-1.amazonaws.com/log-analyzer:0.0.1

The 'Registros' tab is selected, showing a log history with the following entries:

- 07-07-2025 12:33:49 AM [AppRunner] Deployment with ID 5152bc029fd3450ba7bea040909249ec3 started. Triggering event : SERVICE\_CREATE
- 07-07-2025 12:33:49 AM [AppRunner] Deployment Artifact: [Repo Type: ECR], [Image URL: 128624919537.dkr.ecr.us-east-1.amazonaws.com/log-analyzer], [Image Tag: 0.0.1]
- 07-07-2025 12:34:14 AM [AppRunner] Pulling image 128624919537.dkr.ecr.us-east-1.amazonaws.com/log-analyzer from ECR repository.
- 07-07-2025 12:34:15 AM [AppRunner] Successfully pulled your application image from ECR.
- 07-07-2025 12:34:37 AM [AppRunner] Provisioning instances and deploying image for publicly accessible service.
- 07-07-2025 12:34:47 AM [AppRunner] Performing health check on protocol: TCP [Port: '80'].
- 07-07-2025 12:35:46 AM [AppRunner] Health check is successful. Routing traffic to application.

The last entry is highlighted with a red border. At the bottom of the page, there are links for 'CloudShell', 'Comentarios', and legal notices.

## Probando nuestro servicio de App Runner

### Inicio\_Video

Por último, probemos nuestra aplicación. Copia y pega en tu navegador la dirección web indicada bajo la opción “Dominio predeterminado” y añade lo siguiente al final de la misma:

**/logs?start\_time=2021-04-23T10:00:00&end\_time=2027-04-23T10:00:00**

Debería quedar una URL como la siguiente:

**[https://riq7pb5eu.us-east-1.awsapprunner.com/logs?start\\_time=2021-04-23T10:00:00&end\\_time=2027-04-23T10:00:00](https://riq7pb5eu.us-east-1.awsapprunner.com/logs?start_time=2021-04-23T10:00:00&end_time=2027-04-23T10:00:00)**

Tu navegador debería de mostrar el siguiente contenido:

```
{"logs": [{"timestamp": "2023-04-23T10:00:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:01:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:02:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:03:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:04:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:05:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:06:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:07:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:08:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:09:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:10:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:11:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:12:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:13:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:14:00", "tag": "INFO", "message": "Test log"}]}
```

```
log"}, {"timestamp": "2023-04-23T10:15:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:16:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:17:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:18:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:19:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:20:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:22:00", "tag": "ERROR", "message": "Test error"}]]}
```

## Fin\_Video

### Desplegando la aplicación en AWS Lambda – serveless –

Podemos desplegar imágenes de Docker – aplicaciones Dockerizadas – de manera serverless en AWS Lambda. Lo que esto implica, es que AWS será el responsable de la gestión del ciclo de vida del contenedor – inicialización, ejecución, escalado (vertical) y destrucción o eliminación del contenedor.

Aquí tendremos que realizar el mismo paso previo que para desplegar en AWS App Runner, esto siendo, crear un repositorio de ECR. Para este particular caso le asignaremos el nombre lambda-log-analyzer tanto al repositorio de ECR a crear como a la imagen de Docker a construir.

Despleguemos esto!

### ¡Ahora te toca a ti!

¡Es tu turno! Vuelve a revisar los pasos tomados en el despliegue anterior – convertir el Dockerfile en una Docker Image → crear el repositorio de ECR → autenticarse con ECR → etiquetar la imagen de Docker construida → subir la imagen etiquetada a ECR – y adaptarlos para este nuevo.

Ten en cuenta lo siguiente:

- El nombre propuesto tanto para la imagen de Docker como para el repositorio de ECR es lambda-log-analyzer.
- Recuerda taggear tu imagen con una versión como 0.0.1
- Deberás de autenticarte con ECR antes de realizar el docker push.

### Creando un nuevo servicio de AWS Lambda

Una vez creado el repositorio de ECR, sigue los siguientes pasos para crear tu primer servicio de AWS Lambda:

1. Logueate en la interfaz web de AWS, busca por el servicio "Lambda" en la barra de búsqueda.
2. Presiona sobre el servicio mencionado para abrir su panel de opciones.
3. Cliquea el botón "Crear una función".
4. Selecciona la opción "Imagen del contenedor".
5. Bajo el panel "Información básica" asigne un nombre al servicio a crear, para este caso le asigne el nombre: "lambda-log-analyzer-v1".

- Bajo el panel "Información básica", en la opción "URI de imagen de contenedor" presiona el botón "Examinar imágenes".
- Busca el repositorio creado: "lambda-log-analyzer" y selecciona la versión de la imagen subida: 0.0.1. Presiona "Seleccionar imagen".

**Crear una función** Información

Seleccione una de las siguientes opciones para crear la función.

- Crear desde cero**  
Empiece con un sencillo ejemplo "Hello World".
- Utilizar un proyecto**  
Cree una aplicación Lambda utilizando un código de muestra y los ajustes de configuración predefinidos de casos de uso comunes.
- Imagen del contenedor**  
Seleccione una imagen de contenedor para implementar para la función.

**Información básica**

**Nombre de la función**  
Escriba un nombre para describir el propósito de la función.  
**lambda-log-analyzer-v1**

El nombre de la función debe tener entre 1 y 64 caracteres, debe ser exclusivo de la región y no puede incluir espacios. Los caracteres válidos son a-z, A-Z, 0-9, guiones (-) y guiones bajos (\_).

**URI de imagen de contenedor** Información  
La ubicación de la imagen de contenedor que se va a utilizar para la función.  
**128624919537.dkr.ecr.us-east-1.amazonaws.com/lambda-log-analyzer@sha256:efcbb86bb2b62d2cfbea56**

Requiere un URI de Imagen de Amazon ECR válido

**Anulaciones de imágenes de contenedor**

**Arquitectura** Información  
Elija la arquitectura del conjunto de instrucciones que desea para el código de la función.  
 **x86\_64**

**Permisos** Información  
De forma predeterminada, Lambda creará un rol de ejecución con permisos para cargar registros en Amazon CloudWatch Logs. Puede personalizar este rol predeterminado más adelante al agregar los disparadores.

© 2025, Amazon Web Services, Inc. o sus filiales. Privacidad Términos Preferencias de cookies

- Bajo el panel "Configuraciones adicionales" selecciona el valor "Enable" bajo la opción "Habilitar URL de la función".
- Bajo el panel "Tipo de autorización" cambia el valor de "AWS\_IAM" a "NONE".
- Deja lo demás tal cual está y presiona sobre el botón "Crear una función".

**Configuraciones adicionales**

**Networking**

**Habilitar URL de la función** Información  
Utilice URL de función para asignar puntos de enlace HTTPS(S) a la función de Lambda.  
 **Enable**

**Tipo de autorización**  
Elija el tipo de autorización para la URL de la función. [Más información](#)  
 **AWS\_IAM**  
Solo los usuarios y roles de IAM autenticados pueden realizar solicitudes a la URL de la función.  
 **NONE**  
Lambda no realizará la autenticación de IAM en las solicitudes a la URL de la función. El punto de conexión de la URL será público a menos que implemente su propia lógica de autorización en la función.

**Permisos de URL de función**

○ Cuando elige el tipo de autenticación **NONE**, Lambda crea automáticamente la siguiente política basada en recursos y la asocia a la función. Esta política hace que la función sea pública para cualquier persona que tenga la URL de la función. Puede editar la política más adelante. Para limitar el acceso a los usuarios y roles de IAM autenticados, elija el tipo de autenticación **AWS\_IAM**.

**Instrucción de política**

```

1 * []
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "StatementId": "FunctionURLAllowPublicAccess",
6       "Effect": "Allow",
7       "Principal": "*",
8       "Action": "lambda:InvokeFunction",
9       "Resource": "arn:aws:lambda:us-east-1:128624919537:function:lambda-log-analyzer-v1",
10      "Condition": {
11        "StringEquals": {
12          "AWS:SourceArn": "arn:aws:lambda:us-east-1:128624919537:function:lambda-log-analyzer-v1"
13        }
14      }
15    }
16  ]
17 }
  
```

© 2025, Amazon Web Services, Inc. o sus filiales. Privacidad Términos Preferencias de cookies

- Si todo salió bien, el siguiente mensaje debería de aparecer en la interfaz del servicio recién creado.

## Probando nuestro servicio de AWS Lambda

Por último, probemos nuestra aplicación.

### Inicio\_Video

Copia y pega en tu navegador la dirección web indicada bajo la opción “URL de la función” y añade lo siguiente al final de la misma:

**/logs?start\_time=2021-04-23T10:00:00&end\_time=2027-04-23T10:00:00**

Debería de quedar una URL como la siguiente:

**https://lnymcsaopcjton67pm4hxj3ck40lijac.lambda-url.us-east-1.on.aws/logs?start\_time=2021-04-23T10:00:00&end\_time=2027-04-23T10:00:00**

Tu navegador debería de mostrar el siguiente contenido:

```
{"logs": [{"timestamp": "2023-04-23T10:00:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:01:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:02:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:03:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:04:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:05:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:06:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:07:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:08:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:09:00", "tag": "INFO", "message": "Test log"}]}
```

```
log"}, {"timestamp": "2023-04-23T10:10:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:11:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:12:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:13:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:14:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:15:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:16:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:17:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:18:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:19:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:20:00", "tag": "INFO", "message": "Test log"}, {"timestamp": "2023-04-23T10:22:00", "tag": "ERROR", "message": "Test error"}]]}
```

## Fin\_Video

### La pregunta del millón de dólares ¿Cuál es la forma correcta de desplegar esta aplicación?

Para un momento, revisa nuevamente el propósito y problema que resuelve la aplicación, además de las definiciones y funcionamiento de ambos servicios, por un lado uno gestiona todo el ciclo de vida del contenedor – desde su ejecución, hasta su eliminación – y otro solo gestiona la ejecución y el escalado. Piensalo por un momento y trata de responder a la pregunta, justifica tu decisión de por qué crees que es correcta.

La forma correcta y de paso, la mejor forma de desplegar esta aplicación es usando AWS App Runner; veamos porqué.

#### AWS App Runner

AWS App Runner es el servicio adecuado para este caso de uso solo por una razón: nuestra aplicación tiene una base de datos interna que no queremos que sea borrada arbitrariamente, para que esto no pase debemos de escoger un **servicio** que nos **delegue la responsabilidad de la eliminación del contenedor a nosotros** como desarrolladores, de este modo, antes de eliminar el contenedor nos podremos asegurar que los datos recopilados sean guardados en un sistema externo y no se pierdan.

#### AWS Lambda

AWS Lambda no es el mejor servicio para desplegar nuestra aplicación ya que gestiona **TODO el ciclo de vida** de nuestro contenedor, en otras palabras, los datos que se lleguen a almacenar en nuestra base de datos SQLite interna cuando llegue la hora de eliminar el contenedor, serán eliminados con él; perderemos todos los logs almacenados en nuestra base de datos interna cuando AWS Lambda elimine el contenedor.

## Cierre

Bien, en esta clase hemos visto ciertos conceptos importantes sobre las aplicaciones, como desplegarlas, y como Docker y la nube nos ayudan a esta labor, listemos algunas lecciones que nos deja esta clase:

- Comprendimos más a fondo cuales son los problemas al tratar de llevar nuestro software o aplicación a producción y como Docker nos ayuda con ellos.
- Aclaramos el funcionamiento interno de Docker, las diferentes etapas del ciclo de vida y sus componentes - redes, volúmenes, imágenes, contenedores -.
- Entendimos que es el escalamiento vertical - agregar más recursos - y que es el escalamiento horizontal - agregar más instancias de la aplicación - además de los problemas y riesgos que ambos enfoques implican.
- Vimos servicios de AWS como ECR - un almacén o repositorio para imágenes Docker-, App Runner - un servicio para ejecutar imágenes de Docker en donde la responsabilidad por la eliminación del contenedor recae en el desarrollador - y Lambda - un servicio serverless el cual gestiona todo el ciclo de vida del contenedor -.

Nos vemos en una próxima ;)))