



Contenedores



¿Por qué? 🤔

Entornos 🌎

Dependencias y Componentes de una App 📱

Containers 📦

Imagen 🖼

Espera... ¿Es esto es una maquina virtual? 🏞️

Docker 🐳

Imágenes de Docker 📦

Arquitectura de Docker 🏛️

Mi primer container 🎶

Construcción de Imágenes 🚧

Ejemplo 1: Gatitos y más gatitos 🐱

Volúmenes 📁

Eliminar Volúmenes

Compose 🖌️

Estructura General de docker-compose.yml 📄

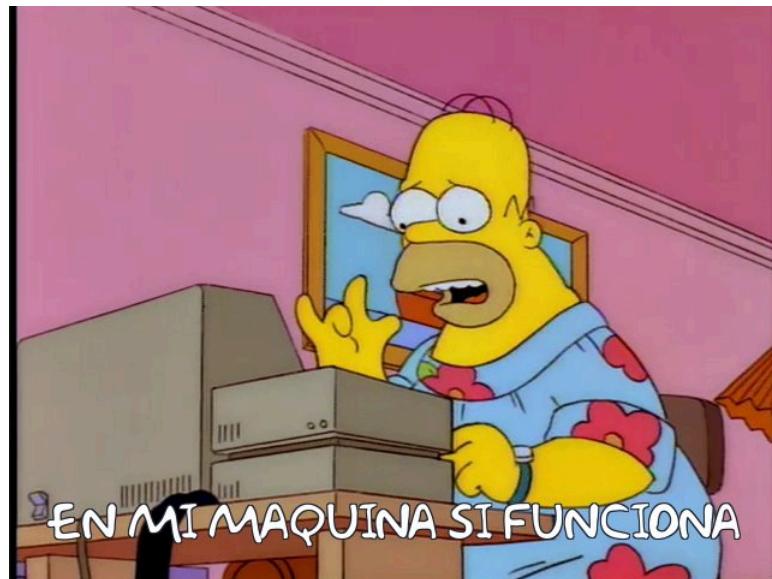
Ejemplo 2: Modelo Simple de Machine Learning 🤖

Conclusión

¿Por qué? 🤔

Uno de los aspectos más cruciales al desarrollar una aplicación para un cliente, o incluso para uso personal, es **garantizar la reproductibilidad del proyecto** en diferentes computadoras. Cuando entregamos un proyecto de programación, debemos asegurarnos de que el desarrollo sea ejecutable en cualquier entorno.

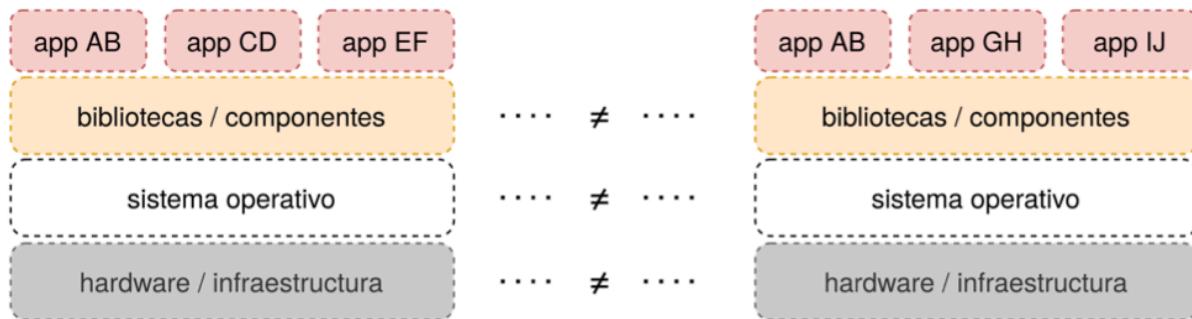
y evitar problemas justificables con frases como: "sorry... en mi computador si corría 😞".



El problema de la irreproducibilidad surge porque el **entorno de desarrollo** en nuestra computadora es **diferente** al de las computadoras de nuestros clientes o de cualquier persona que quiera ejecutar el código. Esto genera problemas de dependencias al ejecutar la aplicación, lo que podría ocasionar errores de ejecución y/o inconsistencias en los resultados obtenidos durante el desarrollo.

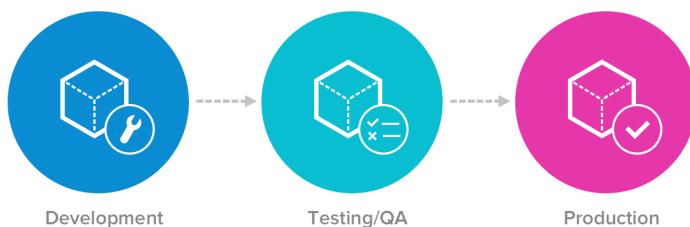
Entornos

En términos prácticos, cuando hablamos de entornos, nos referimos a diferentes máquinas que pueden ser **servidores, computadoras de compañeros, bases de datos, entre otros**. Un punto relevante sobre los entornos en la nube es que, a diferencia de los entornos locales que solemos utilizar, estos **no cuentan con una interfaz gráfica** con la cual podemos interactuar. En su lugar, para optimizar sus requisitos, suelen ser accesibles únicamente mediante línea de comandos.



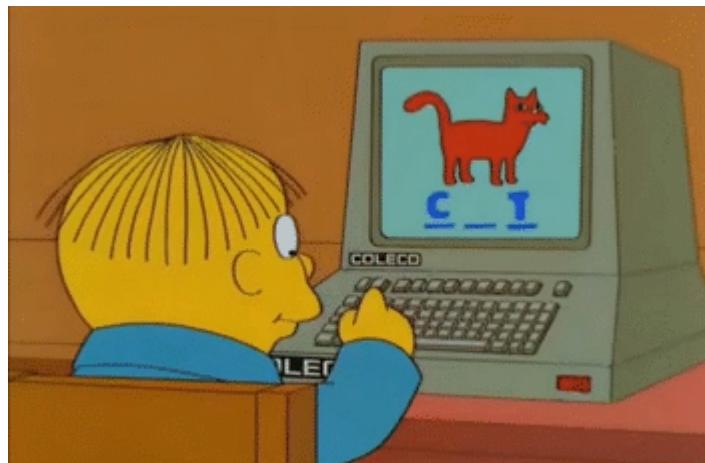
Comparación de componentes en dos entornos.

Al desarrollar un proyecto, generalmente atravesamos tres fases: desarrollo, pruebas (staging) y producción. Una práctica común es que el **desarrollo se realice en máquinas locales**, donde el data-scientist genera un modelo utilizando datos estáticos en su computadora. En **producción**, el entorno ya no es una computadora local, sino un **servicio en la nube** que ejecutará el proceso o modelo creado por el científico de datos. ¿Qué implica esto? Básicamente, el **entorno de producción es completamente diferente al entorno local**.



Las configuraciones y parámetros establecidos en los entornos de producción son diferentes a los utilizados en el desarrollo. Esto se debe a requisitos de seguridad y/o especificaciones de la plataforma. En producción, es común encontrar restricciones adicionales para proteger datos sensibles, implementar autenticación y autorización robustas, y cumplir con normativas de seguridad. Además, las configuraciones de red, almacenamiento y recursos computacionales suelen estar optimizadas para manejar un mayor volumen de tráfico y garantizar alta disponibilidad y escalabilidad.

Dependencias y Componentes de una App



Si pensamos en las dependencias que debe tener un entorno, podríamos suponer que un archivo `requirements.txt` sería suficiente para solucionar el problema de la reproducibilidad, almacenando en este todas las versiones de las librerías utilizadas en el desarrollo y así obtener un resultado similar al que obtuvimos nosotros, entregando la aplicación y/o código. Sin embargo, esto no es del todo cierto 😢.

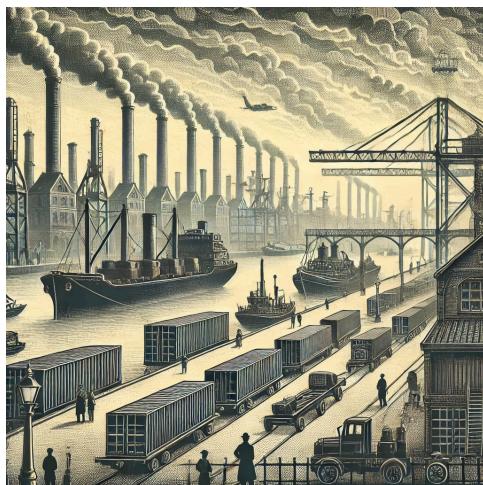
Una aplicación posee más dependencias que solo las librerías utilizadas. Estas dependencias incluyen el sistema operativo, la infraestructura sobre la que se despliega y el hardware subyacente. Revisemos los elementos que conforman una aplicación:

- **Código fuente:** Scripts que forman la base de la aplicación.
- **Librerías:** Todas las bibliotecas y paquetes que la aplicación necesita para funcionar correctamente.
- **Sistema Operativo (SO):** La plataforma en la que se ejecuta la aplicación. Diferencias en las versiones del sistema operativo pueden afectar la compatibilidad y el comportamiento del software.
- **Hardware:** Las especificaciones físicas de la máquina que ejecuta la aplicación, como la CPU, la memoria y el almacenamiento, pueden influir en el rendimiento, estabilidad e inclusive en la reproductibilidad.

Observando los puntos mencionados anteriormente y basándonos en lo que hemos discutido, queda claro que el **hardware es el único elemento que no podemos replicar** exactamente en una computadora en producción. Esto resalta la necesidad de empaquetar los componentes esenciales de una aplicación: el código de la aplicación, las librerías y el sistema operativo. Este proceso nos permite **obtener resultados lo más similares posible a los**

observados durante el desarrollo, minimizando las diferencias entre entornos. Esta práctica se conoce como **Dockerización**.

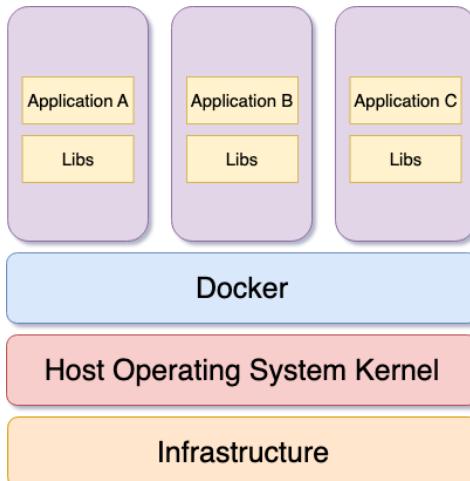
Containers



La contenerización consiste en encapsular o empaquetar el código del software y todas sus dependencias para que pueda ejecutarse de manera uniforme y coherente en cualquier infraestructura. IBM

Por lo tanto, consideramos un **contenedor** como una **unidad de software que incluye todas las dependencias necesarias para su ejecución** y que puede manejar diferentes estados durante su operación. Sus principales características son:

- **Ligeros:** Los contenedores son eficientes en el uso de recursos, permitiendo ejecutar múltiples instancias en un solo sistema sin una sobrecarga significativa.
- **Portables:** Los contenedores pueden moverse fácilmente entre diferentes entornos, ya sea en máquinas locales, servidores on-premise o en la nube.
- **Procesos Aislados:** Los contenedores proporcionan un entorno aislado para cada aplicación, evitando conflictos entre diferentes aplicaciones y mejorando la seguridad.

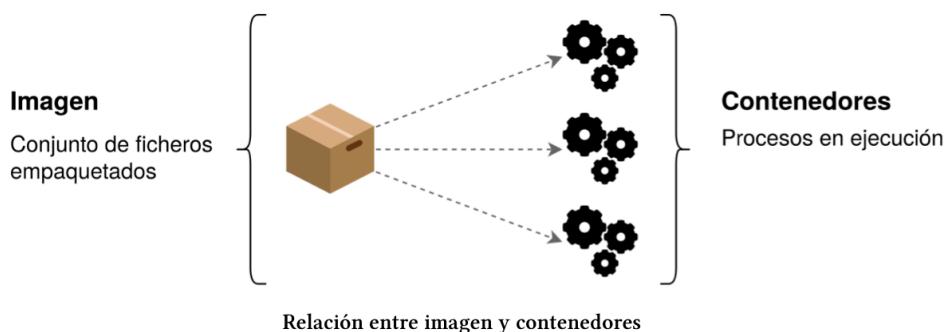


Imagen

Una imagen es un **archivo inmutable que contiene todo lo necesario para ejecutar una aplicación en un contenedor**. Esto incluye el código de la aplicación, las bibliotecas, las dependencias, las herramientas y las configuraciones del entorno.

Es importante notar que **un contenedor no es lo mismo que una imagen**. Una imagen es un conjunto de archivos que incluye el código de la aplicación y sus dependencias, mientras que un contenedor es una instancia en ejecución de esa imagen, gestionando procesos activos.

A una imagen la podemos considerar como el molde que nos permitirá ejecutar multiples containers en paralelos.



Espera... ¿Es esto es una maquina virtual?



Una comparación que se suele hacer con los containers es que la idea se parece mucho a las máquinas virtuales. Si bien, comparten una similitud en la idea, estos mecanismos son muy diferentes entre ellos y a continuación te comentaremos porque:

- **Espacio en Disco**

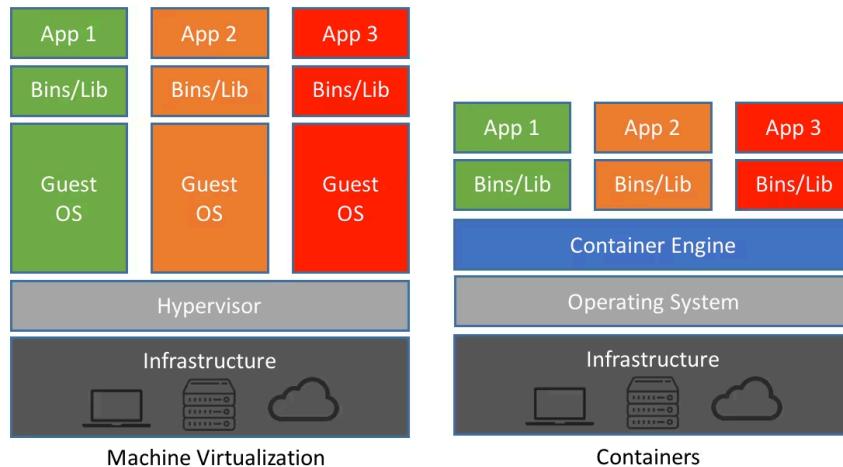
- **Máquinas Virtuales:** Requieren una copia completa del sistema operativo, lo que puede consumir una cantidad significativa de espacio en disco. Cada máquina virtual incluye su propio sistema operativo huésped completo.
- **Contenedores:** Utilizan el sistema operativo del host y comparten sus recursos, lo que reduce drásticamente el espacio en disco necesario. Solo se empaquetan las aplicaciones y sus dependencias específicas, no un sistema operativo completo.

- **Construcción**

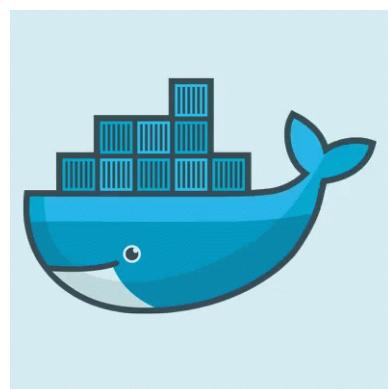
- **Máquinas Virtuales:** Necesitan hipervisores para gestionar las máquinas virtuales, lo que añade una capa adicional de abstracción y complejidad. La creación de una VM implica instalar y configurar un sistema operativo completo.
- **Contenedores:** Son gestionados por el motor de contenedores (como Docker) y se construyen a partir de imágenes ligeras y definidas por archivos de configuración (como veremos más adelante que hace [Dockerfile](#)). Este proceso es más sencillo y rápido, enfocándose en empaquetar únicamente lo necesario para ejecutar la aplicación.

- **Velocidad de Inicio**

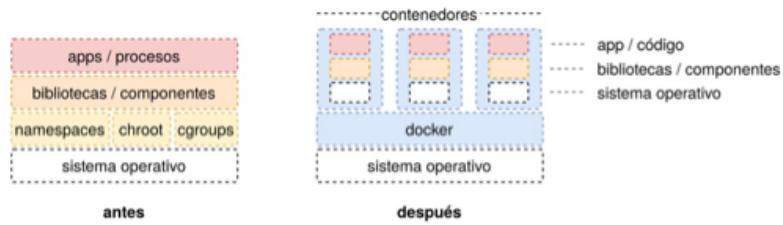
- **Máquinas Virtuales:** Pueden tardar varios minutos en arrancar porque deben inicializar un sistema operativo completo.
- **Contenedores:** Se inician en cuestión de segundos, ya que comparten el núcleo del sistema operativo del host y no necesitan inicializar un sistema operativo completo. Esto permite una escalabilidad y despliegue mucho más rápidos.



Docker es una plataforma que permite **desarrollar, enviar y ejecutar aplicaciones en contenedores**, los cuales son unidades ligeras y portables que incluyen todo lo necesario para que la aplicación funcione, facilitando la consistencia y eficiencia en diferentes entornos. Es importante notar que esta plataforma nos permitirá empaquetar cualquier tipo de aplicación que se encuentre en cualquier idioma de programación, lo cual lo hace una herramienta muy potente y versatil.



Para funcionar, **Docker utiliza componentes del kernel de Linux** para garantizar el aislamiento de los procesos que se desean ejecutar. Entre los procesos más relevantes se encuentran namespaces, que proporcionan aislamiento de recursos del sistema como redes y archivos; chroot, que cambia el directorio raíz de un proceso para crear un entorno aislado; y cgroups, que gestionan y limitan el uso de recursos como CPU y memoria por parte de los contenedores.



Docker encapsula las aplicaciones y sus dependencias en contenedores.

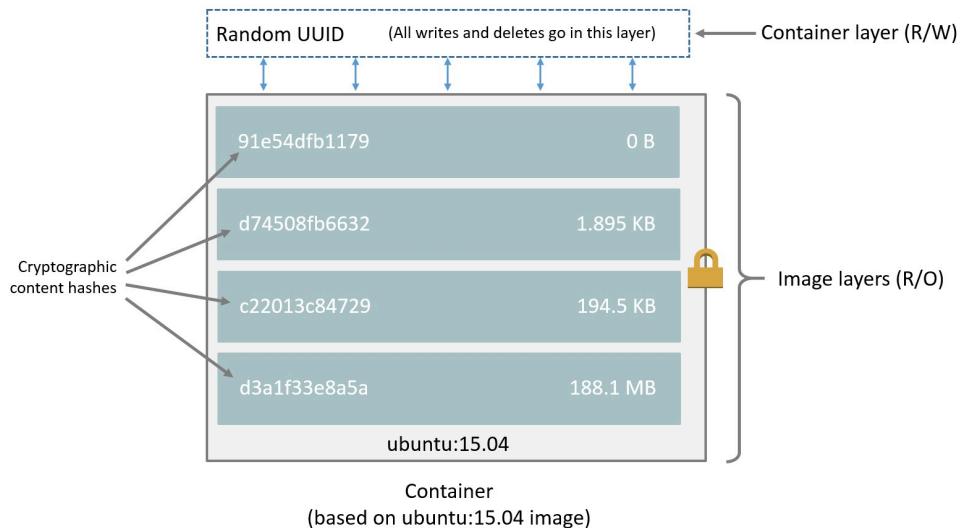
Cada contenedor incluye su propio conjunto de bibliotecas y componentes necesarios para la aplicación, garantizando que se ejecuten de manera aislada y consistente, independientemente del entorno subyacente. Docker se encarga de gestionar estos contenedores, utilizando las capacidades del kernel de Linux para el aislamiento y gestión de recursos, lo que simplifica la implementación y escalabilidad de aplicaciones.



El nombre Docker significa "estibador" en español, un personaje encargado de manejar y organizar las cargas en los barcos.

Imágenes de Docker 📦

Las imágenes de Docker se construyen en capas, lo que significa que cada imagen se compone de una serie de capas apiladas unas sobre otras. Cada capa representa una modificación o adición sobre la capa anterior, creando una estructura de sistema de archivos superpuesta.



Cómo Funcionan las Capas de Docker

- Capa Base:** La primera capa de una imagen de Docker es generalmente una imagen base, como una distribución de Linux (por ejemplo, Ubuntu o Alpine). Esta capa contiene el sistema operativo y las herramientas básicas necesarias.
- Capas Intermedias:** Cada comando en el Dockerfile (el archivo que define cómo se construye la imagen) crea una nueva capa. Por ejemplo, comandos como `RUN apt-get update`, `COPY . /app`, y `ENV PATH=/app/bin:$PATH` generan capas adicionales que se apilan sobre la capa base.
- Capa Superior:** La última capa es la que contiene los cambios finales que hacen que la imagen sea única, como la instalación de la aplicación o configuración específica.

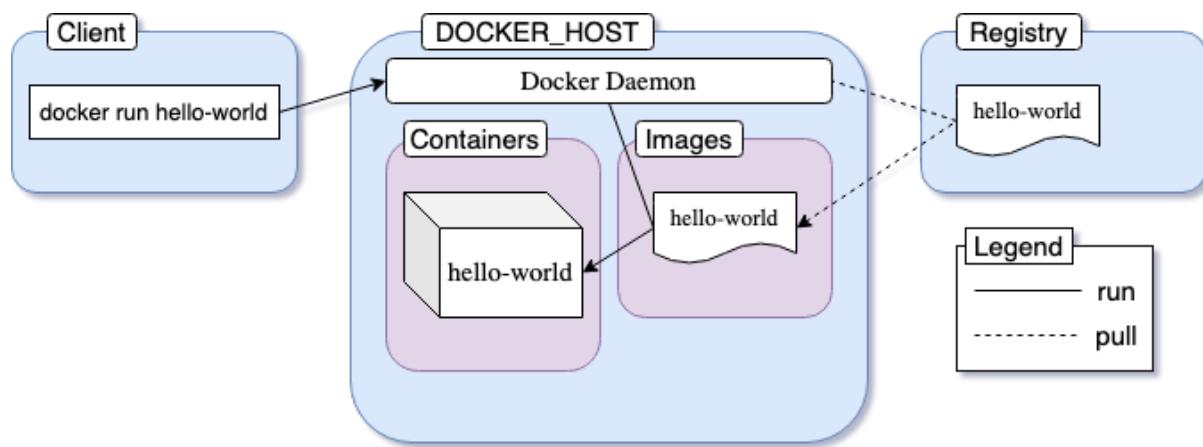
Ventajas de las Imágenes en Capas

- Eficiencia de Almacenamiento:** Las capas se almacenan de manera eficiente porque Docker solo almacena una copia de cada capa. Si múltiples imágenes comparten la misma capa base, Docker no duplica esa capa, lo que ahorra espacio.
- Reutilización de Capas:** Cuando se modifica un Dockerfile y se reconstruye una imagen, Docker solo reconstruye las capas que han cambiado. Las capas que no han cambiado se reutilizan, acelerando el proceso de construcción.
- Portabilidad y Distribución:** Las capas se pueden compartir fácilmente a través de registros como Docker Hub. Al descargar una imagen, solo se

descargan las capas que no están ya presentes en el sistema local, lo que puede acelerar la descarga.

Arquitectura de Docker

La arquitectura de Docker es un conjunto robusto y eficiente de componentes que trabajan juntos para facilitar la creación, despliegue y gestión de aplicaciones en contenedores. En su núcleo, Docker está compuesto por el **Docker Daemon** (dockerd), que se ejecuta en segundo plano en el sistema operativo host y maneja la construcción, ejecución y distribución de contenedores. El **Docker CLI** (Command Line Interface) permite a los usuarios interactuar con el Docker Daemon a través de comandos sencillos.



Otros de los sistemas más relevantes en el ecosistema de Docker son:

- **Trust:** Se encarga de **firmar digitalmente cada una de las imágenes** de los contenedores durante el proceso de construcción, brindando la garantía de que las imágenes utilizadas no han sido interceptadas ni modificadas durante la distribución.
- **Registry:** Permite crear **registros privados de imágenes de contenedores**, facilitando su almacenamiento y distribución segura dentro de una organización.
- **Compose:** Facilita la gestión y orquestación de múltiples contenedores al mismo tiempo, permitiendo **definir y ejecutar aplicaciones multicontenedor con un solo archivo de configuración**.
- **Hub:** Ofrece un registro centralizado para las aplicaciones de Docker, proporcionando **almacenamiento tanto público como privado de**

imágenes de contenedores, y permitiendo compartir y distribuir fácilmente aplicaciones Dockerizadas.

Mi primer container 🎶



Hasta ahora, hemos revisado qué es un contenedor y hemos definido una herramienta que nos permite generarlos fácilmente. Sin embargo, hemos dejado de lado la aplicación práctica. Por eso, este capítulo tiene como objetivo explicar cómo utilizar Docker, revisando algunos comandos relevantes que nos ayudarán en su ejecución.

Para empezar, revisaremos cómo se ejecuta Docker una vez instalado, y exploraremos algunos de los comandos más importantes para su uso. Inicialmente, comenzaremos con el comando de ayuda de Docker, `help`, que proporciona una guía sobre todos los comandos disponibles. Este comando es recomendable utilizarlo como torpedo, no somos maquinas que pueden recordar todo 😅.

```
# Comando de entrada en la consola  
docker --help | less
```

```
# Output esperado en la consola  
A self-sufficient runtime for containers
```

Common Commands:

run	Create and run a new container from an image
exec	Execute a command in a running container
ps	List containers
build	Build an image from a Dockerfile

pull	Download an image from a registry
push	Upload an image to a registry

Del ejemplo podemos destacar **dos aspectos importantes**: primero, Docker nos proporciona una **lista detallada de los comandos disponibles** y, segundo, nos ofrece una **descripción clara de lo que hace cada uno de estos comandos**. Además, es crucial señalar que Docker maneja dos tipos de comandos: `Commands` y `Management Commands`. Los primeros se refieren a comandos básicos y esenciales para el manejo de contenedores, mientras que los `Management Commands` son utilizados para tareas de administración y gestión más avanzadas dentro del ecosistema Docker.

El primer ejemplo de docker que vamos a revisar es el más clásico que se da en todos los lenguajes de programación y este es el "hello world". Para ello, utilizaremos uno de los comandos más básicos de Docker, `run`, que nos permite ejecutar un contenedor a partir de una imagen ya creada.

```
# Comando de entrada en la consola
docker container run hello-world

# Output esperado en la consola
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
478afc919002: Pull complete
Digest: sha256:94323f3e5e09a8b9515d74337010375a456c909543e1ff1538f
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(arm64v8)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

Del comando anterior, podemos analizar el proceso en el que Docker busca y descarga la imagen que se desea ejecutar. Primero, la consola nos notifica que no es posible encontrar la imagen "hello-world" localmente, por lo que procede a descargarla desde Docker Hub:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
```

Un punto relevante sobre el nombre de las imágenes en Docker es que siguen un formato específico que incluye el nombre de la imagen y, opcionalmente, una etiqueta de versión (tag). Por ejemplo, consideremos el siguiente comando: `docker pull homeassistant/amd64-addon-mosquitto:6.4.1`. En este caso, podemos identificar tres partes distintas:

homeassistant/amd64-addon-mosquitto:6.4.1
_____ _____ _____
Repositorio Nombre de la imagen Tag

A continuación, una vez finalizada la descarga, la consola muestra el siguiente mensaje indicando que la descarga se ha completado exitosamente. Podemos observar que el proceso de descarga está asociado a un hash de digest, que asegura la integridad de la imagen descargada:

```
478afc919002: Pull complete
Digest: sha256:94323f3e5e09a8b9515d74337010375a456c909543e1ff153
8f5116d38ab3989
Status: Downloaded newer image for hello-world:latest
```

Este ejemplo demuestra que Docker facilita significativamente la ejecución de contenedores. Con un solo comando, Docker busca la imagen, la descarga si

no está disponible localmente, y la ejecuta, simplificando así el despliegue de aplicaciones.



Si queremos buscar una imagen de manera manual, un comando útil es `search`. Con él, puede buscar imágenes alojadas en Docker Hub desde la consola. Por otro lado, si solo queremos considerar imágenes oficiales, puede usar el siguiente comando:

```
docker search --filter=is-official=true hello-world
```

Ahora que sabemos cómo ejecutar un contenedor, es importante aprender a gestionar diferentes imágenes y contenedores. Lo primero que debemos conocer para realizar una gestión eficaz es qué imágenes tenemos disponibles localmente. Para esto, usaremos el comando `docker images ls` :

```
# Comando de entrada en la consola
docker image ls
```

```
# Output esperado en la consola
REPOSITORY          TAG      IMAGE ID   CREATED    SIZE
hello-world         latest   ee301c921b8a  14 months ago  9.14kB
```

Este comando nos proporcionará una lista de todas las imágenes almacenadas en nuestro sistema local, incluyendo detalles como el repositorio, la etiqueta, el ID de la imagen, la fecha de creación y el tamaño. Con esta información, podemos administrar mejor nuestras imágenes y planificar el uso de contenedores de manera más eficiente.

En el caso de que quisiéramos descargar una imagen en particular, el comando de Docker es `pull`. Este comando permite obtener una imagen específica desde un registro, como Docker Hub, y almacenarla localmente para su uso posterior. Utilicemos el comando en el siguiente ejemplo:

```
# Comando de entrada en la consola
docker pull nginx:1.20.2
docker pull nginx:1.21.6
```

```
# Output esperado en la consola
dc1f00a5d701: Already exists
cfeb1af30280: Pull complete
8b393d1865ed: Pull complete
0e1c4d45c50e: Pull complete
3b5fbcc85c0d: Pull complete
05bbe9de3bca: Pull complete
Digest: sha256:2bcabc23b45489fb0885d69a06ba1d648aeda973fae7bb9
81bafbb884165e514
Status: Downloaded newer image for nginx:1.21.6
docker.io/library/nginx:1.21.6
```

Lo que estamos haciendo aquí es descargar dos imágenes llamadas `nginx`, pero de dos versiones diferentes. Ahora, si volvemos a verificar las imágenes que están en nuestro sistema con el comando `docker image ls`, veremos lo siguiente:

```
# Comando de entrada en la consola
docker image ls

# Output esperado en la consola
REPOSITORY          TAG      IMAGE ID   CREATED    SIZE
nginx               1.21.6   8f05d7383593  2 years ago  134MB
nginx               1.20.2   6d335d16ca83  2 years ago  134MB
hello-world         latest   ee301c921b8a  14 months ago 9.14kB
```

Como podemos observar, acabamos de descargar dos versiones de la misma imagen en nuestro sistema. Ahora, si deseamos eliminar una de ellas, el comando a utilizar es `docker image rm`, donde debemos especificar claramente qué versión queremos eliminar (en este caso, eliminemos la más antigua):

```
# Comando de entrada en la consola (notar como se debe usar el TAG!)
docker image rm nginx:1.20.2

# Output esperado en la consola
Untagged: nginx:1.20.2
Untagged: nginx@sha256:38f8c1d9613f3f42e7969c3b1dd5c3277e635d45
76713e6453c6193e66270a6d
Deleted: sha256:6d335d16ca83d6251c2d53332f0191cfbf8f5f9388ffba854
```

```
87d6a0afea97268
Deleted: sha256:6f04ed6477f7ff662dda99a9316ac14668817fb1fb99d9146
c0a8d3d908f1cb0
Deleted: sha256:172fdb947199b59fdAA4575b3ab043da499121058fb2b6bd
18b8cf258e1ea100
Deleted: sha256:93abdd22aeee57374e76495c34f364e421cdde4c221e32
86841ccf4c84e53d27
Deleted: sha256:d3a85e5b67bc37bb51c5a161ef4f04dee673ea9f4a843be6
8fe1076368cb4510
Deleted: sha256:596cb298af2dc4d9f7d9f55da4d1b37b0e72863dedb2ffbd
3fc65387dacb4c92
Deleted: sha256:62ed8ed20fdbb57a19639fc3a2dc8710dd66cb2364d61ec
02e11cf9b35bc31dc
```

De esta ejecución, podemos ver claramente las múltiples capas que componen la imagen que habíamos descargado. Cada capa está representada por un hash SHA256, lo que demuestra la estructura en capas de las imágenes Docker. Verifiquemos ahora el estado actual de nuestras imágenes:

```
# Comando de entrada en la consola
docker image ls --all

# Output esperado en la consola
REPOSITORY          TAG      IMAGE ID   CREATED    SIZE
hello-world         latest   ee301c921b8a  14 months ago  9.14kB
nginx               1.21.6   8f05d7383593  2 years ago  134MB
```

Con esto, podemos confirmar que efectivamente hemos eliminado la imagen descargada con la versión más antigua, quedándonos con la versión más reciente. Además, como vimos en el ejemplo anterior, al eliminar una imagen, pudimos verificar las capas que la componían. Si deseamos verificar las capas e información más detallada de una imagen, el comando a utilizar es:

```
# Comando de entrada en la consola
docker image inspect nginx:1.21.6 --format '{{json .}}' | jq

# Output esperado en la consola
{
```

```
"Id": "sha256:8f05d73835934b8220e1abd2f157ea4e2260b9c26f6f63a8e3975e7affa46724",
"RepoTags": [
    "nginx:1.21.6"
],
"RepoDigests": [
    "nginx@sha256:2bcabc23b45489fb0885d69a06ba1d648aeda973fae7bb981bafbb884165e514"
],
...
}
```

Otra forma de obtener esta información es revisando el historial de la imagen:

```
# Comando de entrada en la consola
docker history nginx:1.21.6

# Output esperado en la consola
IMAGE      CREATED      CREATED BY          SIZE      COM
MENT
8f05d7383593  2 years ago  /bin/sh -c #(nop) CMD ["nginx" "-g" "daemo
n...  0B
<missing>  2 years ago  /bin/sh -c #(nop) STOPSIGNAL SIGQUIT
0B
<missing>  2 years ago  /bin/sh -c #(nop) EXPOSE 80          0B
<missing>  2 years ago  /bin/sh -c #(nop) ENTRYPOINT ["/docker-ent
r...  0B
<missing>  2 years ago  /bin/sh -c #(nop) COPY file:09a214a3e07c919
a...  4.61kB
<missing>  2 years ago  /bin/sh -c #(nop) COPY file:0fd5fca330dcd6a
7...  1.04kB
<missing>  2 years ago  /bin/sh -c #(nop) COPY file:0b866ff3fc1ef5b0...
1.96kB
<missing>  2 years ago  /bin/sh -c #(nop) COPY file:65504f71f5855ca
0...  1.2kB
<missing>  2 years ago  /bin/sh -c set -x    && addgroup --system -...
60.1MB
<missing>  2 years ago  /bin/sh -c #(nop) ENV PKG_RELEASE=1~bullse
```

```
ye 0B
<missing> 2 years ago /bin/sh -c #(nop) ENV NJS_VERSION=0.7.3
0B
<missing> 2 years ago /bin/sh -c #(nop) ENV NGINX_VERSION=1.21.6
0B
<missing> 2 years ago /bin/sh -c #(nop) LABEL maintainer=NGINX D
o... 0B
<missing> 2 years ago /bin/sh -c #(nop) CMD ["bash"] 0B
<missing> 2 years ago /bin/sh -c #(nop) ADD file:55b4fe3115c684f5
4... 74.3MB
```

Este comando muestra el historial de la imagen especificada, detallando cada capa que la compone, incluyendo información como la instrucción Dockerfile correspondiente, el tamaño y la fecha de creación. Cabe destacar que una capa se considera significativa si genera un cambio de tamaño en la imagen.

Ejecutemos ahora la imagen descargada con comandos de Docker y visualicemos cuántos recursos consume al montar el contenedor:

```
# Comando de entrada en la consola
docker container run nginx:1.21.6

# Comando de entrada en la consola
docker container stats

# Output esperado en la consola
CONTAINER ID  NAME      CPU %     MEM USAGE / LIMIT   MEM %     NE
T I/O    BLOCK I/O    PIDS
99fe98991c0f  my_nginx  0.00%    4.664MiB / 7.667GiB  0.06%    1.25kB
/ 0B  69.6kB / 12.3kB  6
```

De los resultados, podemos observar cuántos recursos está utilizando el contenedor `nginx`. En este caso:

- **CPU %**: El porcentaje de uso del CPU.
- **MEM USAGE / LIMIT**: La cantidad de memoria usada sobre el límite disponible.
- **MEM %**: El porcentaje de uso de la memoria.

- **NET I/O**: El tráfico de red de entrada y salida.
- **BLOCK I/O**: La cantidad de lectura y escritura en el disco.
- **PIDS**: El número de procesos o hilos dentro del contenedor.

Estos datos nos permiten monitorear y gestionar eficientemente los recursos que consumen nuestros contenedores en tiempo real.

Construcción de Imágenes



Hasta ahora, hemos revisado algunos conceptos e interactuado con imágenes y contenedores predefinidos. Sin embargo, **¿qué sucede si queremos crear nuestra propia imagen para ejecutar un proyecto específico?** En este capítulo, exploraremos cómo hacerlo, demostrando que la creación de estas imágenes no es compleja. El objetivo es adaptarse a la estructura que exigen los `Dockerfile` para luego cargarlos como contenedores.

Como mencionamos anteriormente, una imagen es un conjunto de archivos agrupados en capas. Estas imágenes son entidades tangibles que representan archivos en la máquina, conteniendo código y otros elementos de interés. **Los archivos que componen cada capa son generados a través de instrucciones en el `Dockerfile`**. Cada instrucción en un `Dockerfile` es una palabra reservada que se utiliza durante el proceso de construcción de la imagen. Es importante destacar que, aunque cada instrucción realiza una acción, no todas generan archivos. Sin embargo, todas las instrucciones que sí generan archivos crean una capa que Docker registra con un hash único.

Los archivos de la imagen se definen a través de un `Dockerfile`, lo cual ofrece varias **ventajas computacionales y de buenas prácticas**. Entre estas ventajas se incluyen una **mayor claridad**, la posibilidad de incluir un **sistema de control**

de versiones y la **automatización del proceso de construcción**. Gracias al **Dockerfile**, un desarrollador puede identificar rápidamente todas las dependencias y configuraciones necesarias para ejecutar un proyecto.



Como las dependencias del proyecto se definen en el archivo Dockerfile, es crucial no exponer información sensible, como claves de API, tokens u otros datos privados, en él.

En resumen, aprender a crear nuestras propias imágenes mediante **Dockerfile** nos permite personalizar y optimizar el entorno de ejecución de nuestros proyectos, asegurando que se ejecuten de manera consistente y eficiente en cualquier entorno.

Ejemplo 1: Gatitos y más gatitos

Para nuestro primer ejemplo, vamos a crear una aplicación simple que obtenga imágenes de gatitos de forma aleatoria a través de una API y las guarde en una carpeta. Para lograr esto, configuraremos nuestra propia imagen de Docker y crearemos un código sencillo en Python capaz de descargar las imágenes que deseamos.

Empezando con el código, utilizaremos Gradio para generar la API y una función que descargue los gatitos desde el siguiente enlace: [The Cat API](#)

Con Gradio y la API anterior, el código será:

```
import gradio as gr
import requests
from PIL import Image
from io import BytesIO
import os
from datetime import datetime

# Crear una carpeta para almacenar las imágenes de gatos
os.makedirs("saved_cats", exist_ok=True)

def fetch_random_cat_pic():
    try:
        # Enviar una solicitud a The Cat API
```

```

cat_api_response = requests.get("https://api.thecatapi.com/v1/images/se
cat_api_response.raise_for_status()

# Extraer la URL de la imagen de la respuesta de la API
cat_data = cat_api_response.json()
cat_image_url = cat_data[0]['url']

# Descargar la imagen
image_response = requests.get(cat_image_url)
image_response.raise_for_status()

# Abrir la imagen
cat_image = Image.open(BytesIO(image_response.content))

# Generar un nombre de archivo único usando la fecha y hora actual
image_filename = f"cat_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
image_path = os.path.join("saved_cats", image_filename)

# Guardar la imagen localmente
cat_image.save(image_path)

return image_path
except requests.exceptions.RequestException as e:
    print(f"An error occurred: {e}")
    return None

iface = gr.Interface(
    fn=fetch_random_cat_pic,
    inputs=[],
    outputs=gr.Image(type="filepath", label="Random Cat Image"),
    live=True
)

iface.launch()

```

Este código no debería sorprendernos demasiado; sin embargo, un punto muy relevante es la correcta identificación de las librerías necesarias para generar la API. **En este caso, solo necesitamos tres librerías: Gradio, Requests y**

Pillow. Es crucial especificar todas las dependencias de librerías en el Dockerfile para evitar problemas durante la ejecución. Por lo tanto, es recomendable guardar esta información en un archivo `requirements.txt` para facilitar la instalación posterior.

? ¿Qué otro factor debería incluir?

Por otro lado, al definir el archivo Docker, es importante saber que debe contener instrucciones específicas para construir los ficheros. Algunas de las instrucciones más comunes son:

1. FROM:

- **Descripción:** Especifica la imagen base que se utilizará para construir la nueva imagen.
- **Ejemplo:** `FROM python:3.8`

2. LABEL:

- **Descripción:** Añade metadatos a la imagen, como información del autor.
- **Ejemplo:** `LABEL maintainer="tu.email@dominio.com"`

3. RUN:

- **Descripción:** Ejecuta comandos en la imagen durante el proceso de construcción, como la instalación de paquetes necesarios.
- **Ejemplo:** `RUN pip install numpy pandas scikit-learn`

4. COPY:

- **Descripción:** Copia archivos o directorios desde el sistema de archivos del host al sistema de archivos de la imagen.
- **Ejemplo:** `COPY . /app`

5. WORKDIR:

- **Descripción:** Establece el directorio de trabajo para los siguientes comandos.
- **Ejemplo:** `WORKDIR /app`

6. CMD:

- **Descripción:** Proporciona un comando predeterminado que se ejecutará cuando se inicie un contenedor de la imagen.
- **Ejemplo:** `CMD ["python", "train_model.py"]`

7. ENV:

- **Descripción:** Establece variables de entorno en la imagen.
- **Ejemplo:** `ENV PYTHONUNBUFFERED=1`

Estos comandos son suficientes para crear una imagen Docker básica para un proyecto de Machine Learning en general. Luego usando esto tendremos:

```
# Usa una imagen base de Python
FROM python:3.9-slim

# Establece el directorio de trabajo
WORKDIR /app

# Copia el archivo requirements.txt en el contenedor
COPY requirements.txt .

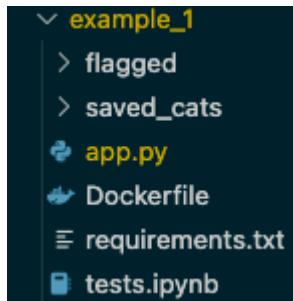
# Instala las dependencias necesarias
RUN pip install --no-cache-dir -r requirements.txt

# Copia el resto de la aplicación en el contenedor
COPY ..

# Exponer el puerto que utiliza Gradio (por defecto 7860)
EXPOSE 7860

# Comando para ejecutar la aplicación
CMD ["python", "app.py"]
```

Este Dockerfile instala las bibliotecas necesarias de Python, copia el código de tu proyecto al contenedor y establece el comando para ejecutar el script principal de la API de gatitos. En base a cada uno de los archivos expuestos, tendremos un directorio con la siguiente estructura:

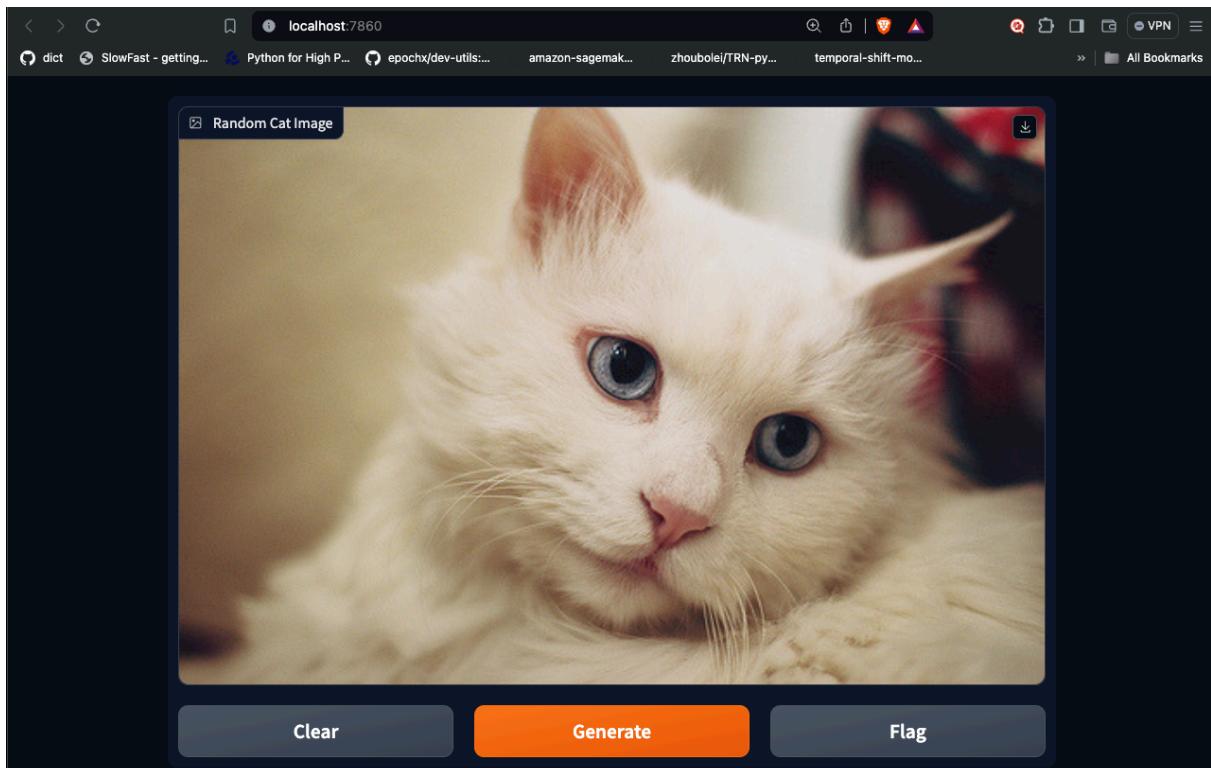


Es importante notar que, debido a que se realizaron pruebas previas a la creación del código en un notebook, se creó una carpeta de gatos dentro de la carpeta a dockerizar. ¿Tendrá esto un impacto?

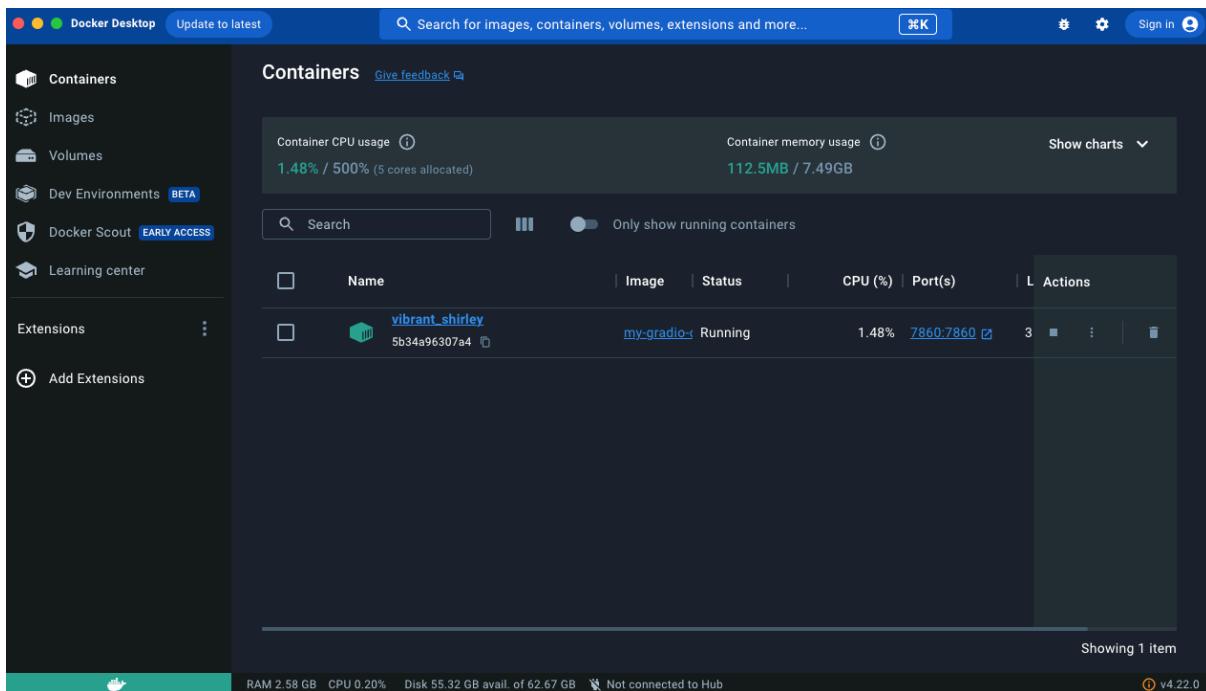
Finalmente, el código en consola para crear la imagen y ejecutar el contenedor será:

```
docker build -t my-gradio-cat-app . # comando para generar la imagen  
docker run -p 7860:7860 my-gradio-cat-app # comando para ejecutar la imaq
```

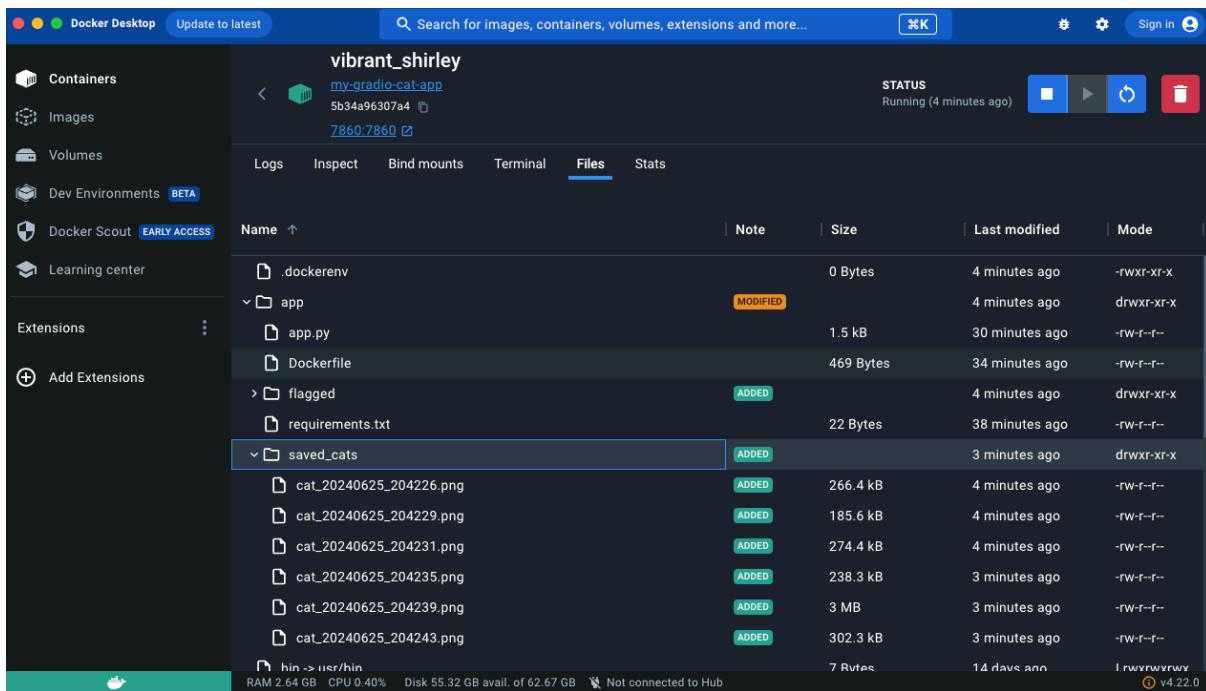
Abrimos el puerto en el que esta alojado la API:



Es recomendable que una vez que hallan lanzado esta API naveguen en la interfaz de cliente que nos ofrece Docker, de esta forma podrán profundizar con mas detalle los beneficios que nos ofrece docker:



En particular, un factor interesante a revisar es **cómo se guardan los archivos producidos por nuestra API dentro del contenedor Docker**. Estos **archivos quedan alojados en el contenedor**, lo cual puede generar una complicación si deseamos acceder a ellos desde fuera del contenedor o si necesitamos persistencia de datos a largo plazo.



 ¿Como podríamos solucionar esto?

Volúmenes

Los volúmenes de Docker son una forma de almacenar datos que persisten incluso después de que el contenedor ha sido eliminado. **Permiten que los contenedores lean y escriban datos directamente en el sistema de archivos del host.** Esto es útil para la persistencia de datos, el intercambio de datos entre contenedores y para mantener los datos fuera del contenedor, lo cual facilita las copias de seguridad y la gestión de datos.

Entre las **características** que poseen los volúmenes tenemos

- **Persistencia:** Los datos almacenados en un volumen persisten más allá del ciclo de vida del contenedor, lo que significa que no se pierden cuando el contenedor se detiene o se elimina.
- **Desempeño:** Los volúmenes son gestionados por Docker y están optimizados para el rendimiento.
- **Compatibilidad entre plataformas:** Los volúmenes funcionan de la misma manera en diferentes sistemas operativos, proporcionando una forma coherente de manejar el almacenamiento.
- **Backups y restauraciones:** Los volúmenes son fáciles de respaldar y restaurar, ya que están ubicados en el sistema de archivos del host.
- **Compatibilidad entre contenedores:** Varios contenedores pueden compartir el mismo volumen, lo que facilita el intercambio de datos.

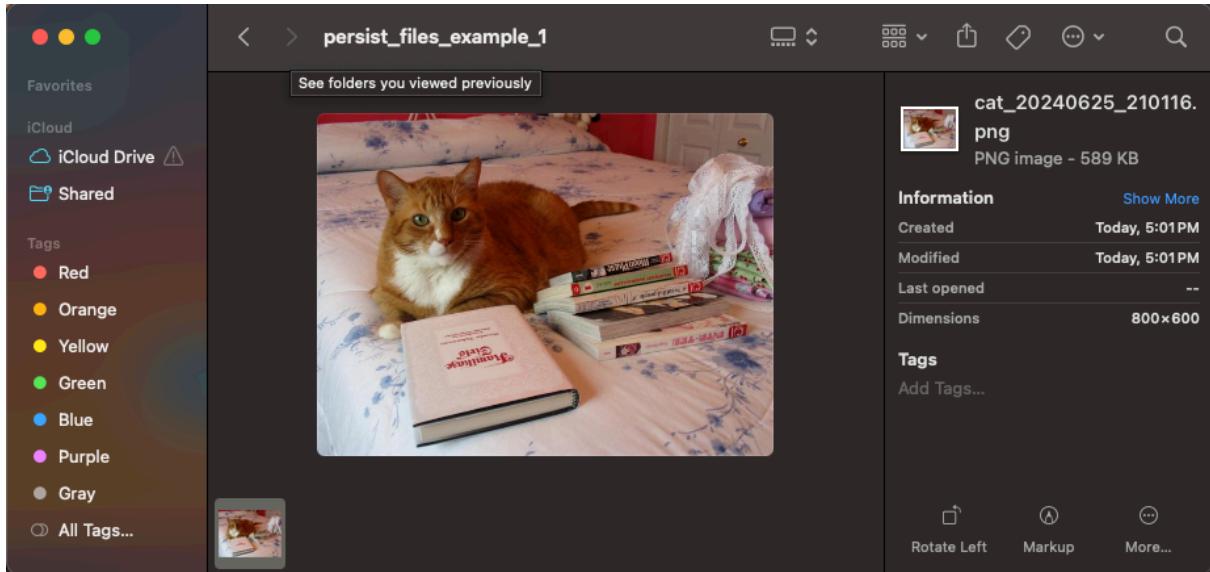
Retomando el problema anterior, podemos utilizar volúmenes de Docker, que permiten que los datos generados en el contenedor se almacenen en el sistema de archivos del host.

Por ejemplo, podemos modificar el comando de ejecución del contenedor para montar un volumen que enlace una carpeta del host con una carpeta del contenedor donde se guardan las imágenes:

```
docker run \
-p 7860:7860 \
```

```
-v <directorio_local>:<directorio_container> \
my-gradio-cat-app # ejecutar como volumen
```

Veamos ahora qué sucede al utilizar los volúmenes. Si revisamos la carpeta, encontraremos que efectivamente un gatito lector nos está esperando allí:



De esta manera, las imágenes de gatitos generadas por la API se almacenarán en la ruta especificada en el host, permitiendo un fácil acceso y persistencia.



Existen varias formas de montar un volumen. Una alternativa a la que vimos es utilizar la opción `--mount`, que permite especificar diferentes métodos para montar archivos. Por ejemplo, `volume` se usa comúnmente para la persistencia de datos, mientras que `bind` se utiliza para señalar archivos de configuración. Aquí tienes un ejemplo de uso:

```
--mount type=volume,src=a,dst=b
```

Eliminar Volúmenes

Visualizamos la importancia que tenía la persistencia de los datos con el ejemplo anterior, pero con esto aparece la pregunta: ¿cómo podemos recuperar el espacio que utilizamos en los volúmenes?. Realizar esto es simple y para ello vamos utilizar el siguiente comando:

```
# Comando de entrada en la consola
docker system df

# Output esperado en la consola
TYPE      TOTAL ACTIVE SIZE RECLAMABLE
Images     9      2    2.437GB 2.071GB (84%)
Containers 3      0    12.17MB 12.17MB (100%)
Local Volumes 7      0    70.02MB 70.02MB (100%)
Build Cache 23      0    202.2kB 202.2kB
```

el resultado, nos va a interesar principalmente la sección de "Local Volumes", que nos indica la cantidad de volúmenes locales que tenemos, su tamaño total y cuánto de ese espacio es recuperable.

```
# Comando para eliminar volúmenes no utilizados
docker volume prune
```

Por otro lado, si queremos especificar que volumen queremos eliminar y queremos ser específicos en este, el comando sería:

```
# Comando para eliminar volúmenes no utilizados
docker volume rm [volumen-name]
```

Compose 🖌



Docker Compose es una herramienta que permite definir y ejecutar aplicaciones Docker multi-contenedor. Con Compose, puedes usar un archivo

YAML para configurar los servicios de tu aplicación. Luego, con un solo comando, puedes crear e iniciar todos los servicios desde tu configuración.

Ventajas de Docker Compose

- **Definición simple:** Utiliza un solo archivo YAML para definir múltiples servicios.
- **Orquestación:** Gestiona la creación y el ciclo de vida de los contenedores de manera eficiente.
- **Aislamiento:** Mantiene la consistencia entre entornos de desarrollo, prueba y producción.

El archivo `docker-compose.yml` es el núcleo de Docker Compose, donde se define la configuración de los servicios que componen una aplicación Docker multi-contenedor. Vamos a desglosar su estructura y contenido detalladamente para entender qué hace cada parte.

Estructura General de `docker-compose.yml`

El archivo `docker-compose.yml` utiliza el formato YAML para definir los servicios, redes y volúmenes. A continuación, se explica una configuración básica:

```
services:  
  web: # contenedor 1  
    image: nginx:latest # imagen 1  
    ports:  
      - "80:80"  
    volumes:  
      - ./html:/usr/share/nginx/html  
  db: # contenedor 2  
    image: mysql:5.7 # imagen 2  
    environment:  
      MYSQL_ROOT_PASSWORD: example  
      MYSQL_DATABASE: exampledb
```

Por otro lado, viendo en detalle cada uno de los puntos que posee este archivo tenemos:

- Definición de Servicios

services:

La clave `services` define los servicios que componen la aplicación. Cada servicio se ejecuta en un contenedor separado y puede depender de otros servicios.

- Servicio `web`

web:

```
image: nginx:latest
ports:
  - "80:80"
volumes:
  - ./html:/usr/share/nginx/html
```

- `web`: Es el nombre del servicio. Puedes elegir cualquier nombre que sea significativo para tu aplicación.
- `image: nginx:latest`: Especifica la imagen de Docker que se utilizará para el contenedor. En este caso, se está usando la última versión de la imagen oficial de Nginx.
- `ports: - "80:80"`: Mapea el puerto 80 del contenedor al puerto 80 del host. Esto permite acceder al servidor web Nginx desde el host a través del puerto 80.
- `volumes: - ./html:/usr/share/nginx/html`: Monta el directorio `./html` del host en el directorio `/usr/share/nginx/html` del contenedor. Esto permite servir archivos HTML desde el host a través del contenedor Nginx.

- Servicio `db`

db:

```
image: mysql:5.7
environment:
  MYSQL_ROOT_PASSWORD: example
  MYSQL_DATABASE: exampledb
```

- `db`: Es el nombre del servicio de la base de datos.
- `image: mysql:5.7`: Especifica la imagen de Docker para el contenedor MySQL, en este caso, la versión 5.7 de MySQL.

- `environment`: Define **variables de entorno** que se pasarán al contenedor. En este caso:
 - `MYSQL_ROOT_PASSWORD: example`: Establece la contraseña del usuario root de MySQL.
 - `MYSQL_DATABASE: exampledb`: Crea una base de datos llamada `exampledb` al iniciar el contenedor. Para ejecutar los servicios definidos en `docker-compose.yml`, se utiliza el comando:

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

Este comando realiza las siguientes acciones:

- 1. Descarga las Imágenes:** Si las imágenes especificadas no están presentes en el sistema local, Docker las descargará del registro de Docker (Docker Hub u otro registro configurado).
- 2. Crea y Inicia los Contenedores:** Crea los contenedores para cada servicio definido en el archivo y los inicia.
- 3. Configura las Redes y Volúmenes:** Configura las redes y volúmenes según lo especificado en el archivo.

Para detener y eliminar los contenedores, se utiliza:

```
docker compose down
```

Este comando detiene y elimina los contenedores, redes y volúmenes creados por `docker compose up`.

Ejemplo 2: Modelo Simple de Machine Learning

En este ejemplo, abordaremos un problema típico de machine learning utilizando un contenedor en Docker. Es crucial entender que, generalmente, cuando se realiza el despliegue de un modelo de machine learning, estos se implementan a través de objetos binarios de Python que son cargados por un sistema de contenedores.

Primero, comenzamos con el desarrollo del modelo. Simplificaremos el problema de clasificación a la etapa de creación del modelo utilizando la librería `scikit-learn` y el dataset de iris. Con el modelo entrenado, es importante guardarlo junto con los estimadores obtenidos durante el entrenamiento, ya

que queremos aplicar este modelo en un entorno que más tarde será el productivo. Para esto, podemos usar las librerías `pickle` o `joblib` de Python. De esta forma, el archivo quedará binarizado en la memoria local de su computadora y podrá ser referenciado fácilmente para su utilización.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import joblib

iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)

joblib.dump(clf, 'iris_classifier.joblib')
```

A continuación, vamos a crear la aplicación que se encargará de ejecutar el modelo en nuestro contenedor. Para esto, utilizaremos `FastAPI`, y nuestro objetivo será que cada vez que se ejecute el contenedor, se guarden las predicciones en un archivo local (para lo cual utilizaremos volúmenes más adelante).

```
from fastapi import FastAPI
from pydantic import BaseModel
from joblib import load
import os
import pandas as pd

class IrisData(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float
```

```

model = load("iris_classifier.joblib")
app = FastAPI()

data_dir = os.getenv('DATA_DIR', '/data')
csv_file_path = os.path.join(data_dir, "predictions.csv")
if not os.path.exists(csv_file_path):
    pd.DataFrame(columns=["sepal_length", "sepal_width", "petal_length", "petal_width"]).to_csv(csv_file_path)

@app.get("/")
def read_root():
    return {"message": "Iris Classifier API is running!"}

@app.post("/predict")
def predict(data: IrisData):
    features = [[data.sepal_length, data.sepal_width, data.petal_length, data.petal_width]]
    prediction = model.predict(features)[0]

    new_data = pd.DataFrame([
        "sepal_length": data.sepal_length,
        "sepal_width": data.sepal_width,
        "petal_length": data.petal_length,
        "petal_width": data.petal_width,
        "prediction": prediction
    ])
    new_data.to_csv(csv_file_path, mode='a', header=False, index=False)

    return {"prediction": int(prediction)}

```

Luego generamos nuestra imagen que contendrá el código de predicción:

```

FROM python:3.9-slim

WORKDIR /app

COPY app/requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

```

```
COPY app /app

RUN mkdir -p /data

ENV DATA_DIR=/data

EXPOSE 80

CMD ["unicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```

De forma similar, haremos lo mismo con una app de Gradio como frontend. Lo primero es generar la función de predicción para conectar con el backend:

```
import requests
import os

# referencia a localhost si se ejecuta localmente, usa la variable de ambiente I
url_base = os.getenv("BACKEND_URL", "http://localhost:8000") # url del backe

# primero definimos una función para obtener respuestas del back a través de
def get_backend_prediction(
    sepal_length: float,
    sepal_width: float,
    petal_length: float,
    petal_width: float,
):

    # payload
    data = {
        "sepal_length": sepal_length,
        "sepal_width": sepal_width,
        "petal_length": petal_length,
        "petal_width": petal_width,
    }

    url = url_base + "/predict" # url del back end
    response = requests.post(url, json = data) # código de respuesta
```

```
label = response.json().get("label") # obtener contenido de la respuesta  
  
return label
```

Noten como se usa la variable de ambiente `BACKEND_URL`, la cual nos ayudará a hacer referencia al *backend* en caso de que se ejecute el código desde un contenedor (notar que esta variable se crea desde el Dockerfile).

Con esto, la aplicación de Gradio queda definida por el siguiente código:

```
import gradio as gr  
from utils import get_backend_prediction  
import requests  
  
# primero definimos una función para obtener respuestas del back a través de  
def get_backend_prediction(  
    sepal_length: float,  
    sepal_width: float,  
    petal_length: float,  
    petal_width: float,  
):  
  
    # payload  
    data = {  
        "sepal_length": sepal_length,  
        "sepal_width": sepal_width,  
        "petal_length": petal_length,  
        "petal_width": petal_width,  
    }  
  
    url = "http://127.0.0.1:8000/predict" # url del back end  
    response = requests.post(url, json = data) # código de respuesta  
    label = response.json().get("prediction") # obtener contenido de la respuesta  
  
    return label  
  
# la única linea que cambia es la función a invocar  
with gr.Blocks(theme = gr.themes.Base()) as demo:  
    gr.Markdown(
```

```

"""
# Iris ML Demo
Bienvenid@ a Iris ML demo! Esta herramienta esta diseñada para predecir la
## Cómo usar este demo?
Usar esta herramienta es fácil! Sólo debes seguir los siguientes pasos:
1. Fijar los valores de **Sepal Length**, **Sepal Width**, **Petal Length** y
2. Observar el tipo de flor que predice el modelo.

Eso es todo! Estás list@ para explorar y predecir diferentes tipos de flores.
""")

with gr.Row():
    with gr.Column():
        sepal_length_slider = gr.Slider(label = 'Sepal Length', minimum = 0, maximum = 10)
        sepal_width_slider = gr.Slider(label = 'Sepal Width', minimum = 0, maximum = 4)
        petal_length_slider = gr.Slider(label = 'Petal Length', minimum = 0, maximum = 7)
        petal_width_slider = gr.Slider(label = 'Petal Width', minimum = 0, maximum = 2)

    with gr.Column():
        label = gr.Text(label = 'Predicted Label') # se define un nombre para la variable

with gr.Row():
    button = gr.Button(value = 'Predict!')

# setear interactividad
inputs = [sepal_length_slider, sepal_width_slider, petal_length_slider, petal_width_slider]
outputs = [label]

# obtener predicción desde el backend
button.click(fn = get_backend_prediction, inputs = inputs, outputs = outputs)

examples = [
    [0.5, 1.5, 2.5, 3.5], # example 1
    [1, 3, 5, 7], # example 2
]
gr.Examples(examples = examples, inputs = inputs)

demo.launch(server_name="0.0.0.0", server_port=7860, share = True)

```

Por último, creamos un Dockerfile para levantar el frontend:

```
# Usa una imagen base de Python
FROM python:3.9-slim

# Establece el directorio de trabajo
WORKDIR /app

# Copia el archivo requirements.txt en el contenedor
COPY requirements.txt .

# Instala las dependencias necesarias
RUN pip install --no-cache-dir -r requirements.txt

# Copia el resto de la aplicación en el contenedor
COPY ..

# Crear variable de ambiente para referenciar al backend
ENV BACKEND_URL=http://backend:8000

# Exponer el puerto que utiliza Gradio (por defecto 7860)
EXPOSE 7860

# Comando para ejecutar la aplicación
CMD ["python", "app.py"]
```

Ya con el backend y frontend desarrollado, podemos al fin conectarlos todo desde Docker Compose. Para esto, crearemos una configuración para especificar el mapeo de puertos y los volúmenes que utilizaremos. Es importante notar que este archivo nos facilitará la configuración al levantar un contenedor, especialmente cuando trabajamos con aplicaciones más complejas, donde la configuración manual se vuelve más tediosa.

```
services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
```

```
volumes:  
- ./persist_files_example_2:/data
```

```
frontend:  
build: ./frontend  
ports:  
- "7860:7860"
```

De esto, nuestra carpeta deberá lucir de la siguiente forma:

```
└── example_2  
    ├── backend  
    │   ├── data  
    │   └── .dockerignore  
    │   └── Dockerfile  
    │   └── iris_classifier.joblib  
    │   └── main.py  
    └── model_creation.ipynb  
        └── requirements.txt  
    └── frontend  
        ├── .dockerignore  
        ├── app.py  
        └── Dockerfile  
            └── requirements.txt  
        └── utils.py  
    └── docker-compose.yml
```

El paso final será diferente a lo que estamos acostumbrados debido a que hora no levantaremos un container en particular, sino que haremos un compose, para esto debemos utilizar:

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

Al revisar la ejecución, podemos darnos cuenta su ejecución es exitosa si podemos ver el siguiente mensaje en el navegador:

Iris ML Demo

Bienvenid@ a Iris ML demo! Esta herramienta esta diseñada para predecir la clase de una flor a partir de sus características usando Machine Learning.

Cómo usar este demo?

Usar esta herramienta es fácil! Sólo debes seguir los siguientes pasos:

1. Fijar los valores de Sepal Length, Sepal Width, Petal Length y Petal Width.
2. Observar el tipo de flor que predice el modelo.

Eso es todo! Estás list@ para explorar y predecir diferentes tipos de flores. Que lo disfrutes!

Sepal Length	Sepal Width	Petal Length	Petal Width
0.5	1.5	2.5	3.5
1	3	5	7

Es usual que al trabajar con docker compose nos encontremos con *bugs* o errores en el funcionamiento de nuestra aplicación. Una excelente forma para resolver esto es ingresando a la terminal de nuestro contenedor y debuggear directamente. Podemos hacer esto a través de:

```
docker compose exec <nombre_servicio> bash
```

Conclusión

De la clase pudimos revisar cómo Docker garantiza la reproducibilidad de aplicaciones en diferentes entornos, resolviendo problemas de dependencias y diferencias entre entornos de desarrollo y producción. Con esto, vimos cómo crear y gestionar contenedores e imágenes Docker, y comprendieron las ventajas de Docker sobre las máquinas virtuales en términos de eficiencia y portabilidad. Además, exploramos Docker Compose para orquestar aplicaciones multi-contenedor y se realizaron ejemplos prácticos de dos aplicaciones de machine learning con `scikit-learn`, `gradio` y `FastAPI`.

