Docker taller

Introducción

Docker → construccion de imagenes → facilitamos el compartir el codigo a otra persona con las dependencias y todo lo necesario para poder trabajar

Docker es una plataforma de virtualización a nivel de sistema operativo que permite crear, distribuir y ejecutar aplicaciones en contenedores de software controlados. Un contenedor es una unidad estándar de software que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de manera rápida y confiable en diferentes entornos informáticos, asegurando la consistencia, portabilidad y eficiencia en su despliegue y ejecución. Es decir, un contenedor permite que una aplicación funcione de la misma manera sin importar dónde se ejecute, ya sea en una computadora local, en un servidor o en la nube, evitando problemas de compatibilidad y facilitando su gestión y escalabilidad.

Las funciones principales de Docker son:

- **Containerización:** Empaqueta aplicaciones y sus dependencias en contenedores aislados
- **Portabilidad:** Los contenedores pueden ejecutarse en cualquier sistema que tenga Docker instalado, independientemente del sistema operativo subyacente
- **Eficiencia:** Los contenedores comparten el kernel del sistema operativo host, lo que los hace más ligeros que las máquinas virtuales tradicionales
- **Escalabilidad:** Facilita la creación y gestión de múltiples instancias de una aplicación
- **Consistencia:** Garantiza que la aplicación se ejecute de la misma manera en diferentes entornos (desarrollo, pruebas, producción)
- usos de docker
  - facilita la creación de entornos de desarrollo para todo tipo de proyectos
  - despliegue de aplicaciones → de forma rápida en diferentes entornos de ejecución, sin importar los distintos sistemas operativos
  - PERMITE LA EJECUCIÓN DE PROGRAMAS DE MANERA AISLADA → COMO SI FUERA UNA MAQUINA VIRTUAL

Docker utiliza un sistema de imágenes y contenedores donde: (los conceptos los explicaremos detalladamente más adelante)

- **Imágenes:** Son plantillas de solo lectura que contienen las instrucciones para crear un contenedor
- **Contenedores:** Son instancias ejecutables de las imágenes que pueden iniciarse, detenerse, moverse y eliminarse

Conceptos principales

imagenes → representacion estática de una aplicación. se conserva el estado de la aplicación en un tiempo determinado con los cambios que se han realizado la aplicación.

- importante → si realizamos una imagen de la aplicación y después realizamos cambios ⇒ no se van a actualizar en la imagen, tendriamos que volver a hacer una img para coger esos ultimos cambios

> las imagenes son necesarias para la creación de los contenedores. Funcionan como si fueran una receta o los planos necesarios para construir una casa.

contenedores → son como “cajas” virtuales donde se van a empaquetar una aplicación y todas las cosas (como librerias, dependencias, configuraciones..) que la permiten funcionar

despliegue

almacenamiento

COMO CREAR UN CONTENEDOR DOCKER

- **Dockerfile** → Se usa para crear una imagen Docker.
- **Imagen Docker** → Se genera a partir de un Dockerfile y contiene todo lo necesario para ejecutar la aplicación.
- **Contenedor** → Es una instancia en ejecución de una imagen Docker.

Dockerfile

Un Dockerfile es un archivo de texto que contiene un conjunto de instrucciones que le indican a Docker cómo construir una imagen Docker. Es una especie de "receta" que describe paso a paso cómo debe crearse la imagen, especificando qué sistema base utilizar, qué dependencias instalar, qué archivos copiar, qué comandos ejecutar y cómo configurar el entorno, todo con el objetivo de definir el comportamiento de la imagen cuando se ejecute.

Es decir, un Dockerfile es un archivo donde defines todo lo necesario para crear una imagen Docker.

- Estructura básica de un Dockerfile Un Dockerfile consta de una serie de instrucciones que se ejecutan de forma secuencial. Algunas de las instrucciones más comunes son:
  1. `FROM` especifica la imagen base a partir de la cual se construye la nueva imagen. Por ejemplo, `FROM node:14` . Es la primera línea del Docker.
  2. `**RUN**` te permite **configurar** e **instalar cosas** dentro de la imagen **mientras se está construyendo**. Por ejemplo para instalar alguna dependencia o actualizar el sistema.

3. `COPY` copia archivos o directorios desde el sistema de archivos local (tu computadora) a la imagen. (Se usa mucho para copiar archivos de tu pagina web o programa de tu máquina local a la imagen)
4. `ADD` similar a `COPY`, pero también tiene la capacidad de descargar archivos desde una URL y descomprimir archivos tar. (también se usa para archivos de tu pagina web o programa de tu máquina local a la imagen)
5. `WORKDIR`: Establece el directorio de trabajo para el contenedor.
6. `CMD` o `ENTRYPOINT`: Define el comando que se ejecutará cuando se inicie la imagen.
7. `EXPOSE`: Informa a Docker sobre los puertos que se van a utilizar, pero no los abre realmente.
8. `ENV`: Define variables de entorno que estarán disponibles en el contenedor durante su ejecución.
9. `**VOLUME**` te permite crear un directorio persistente/permanente fuera del contenedor para almacenar datos, lo que asegura que los datos no se pierdan incluso si el contenedor se elimina o se reinicia.

Ejemplo de Dockerfile —

Important

#Usa una imagen base de Node.js

```
FROM node:14
```

#Establece una variable de entorno

```
ENV DB_HOST=localhost
```

#Actualiza la lista de paquetes e instala curl

```
RUN apt-get update && apt-get install -y curl
```

#Establece el directorio de trabajo dentro del contenedor

```
WORKDIR /app
```

#Copia el archivo de aplicación al contenedor

```
COPY . /app
```

#Usa ADD para descargar y descomprimir un archivo tar

```
ADD [https://example.com/sample-file.tar.gz](https://example.com/sample-file.tar.gz) /app/files/
```

#Instala las dependencias de la aplicación

```
RUN npm install

#Expone el puerto en el que la aplicación se ejecutará

EXPOSE 3000

#Define un volumen para persistir datos

VOLUME ["/data"]

#Comando por defecto cuando se ejecute el contenedor

CMD ["npm", "start"]
```

Este archivo describe cómo construir una imagen que incluye:

- La imagen base de Node.js.
- El directorio de trabajo donde se ejecutarán los comandos.
- Los pasos para copiar los archivos, instalar dependencias y arrancar la aplicación.

El resultado de ejecutar las instrucciones de un Dockerfile es una Imagen. Un paquete inmutable que contiene todo lo necesario para ejecutar la aplicación: el sistema operativo, el servidor web, el código de la aplicación, las dependencias, las configuraciones, etc.

Imagen Docker

Una imagen Docker es un **conjunto de archivos que incluye todo lo necesario para que una aplicación funcione correctamente, como el código, las herramientas, las librerías y la configuración.**

Estas imágenes no cambian y están organizadas en diferentes **capas**, lo que hace que ocupen menos espacio y se puedan reutilizar fácilmente. A partir de una imagen, se pueden crear contenedores, que son versiones activas de la aplicación listas para usarse.

Las imágenes se pueden guardar y compartir en plataformas como Docker Hub, lo que permite usarlas en diferentes computadoras o servidores sin preocuparse por problemas de compatibilidad.

Por lo que básicamente, una imagen Docker es como una plantilla o "molde" que tiene todo lo necesario para que una aplicación funcione en cualquier lugar.

Una imagen Docker suele contener:

- El sistema operativo base (por ejemplo, Ubuntu, Alpine, etc.).
- Aplicaciones y dependencias necesarias para ejecutar el software.
- Variables de entorno y configuraciones.
- Herramientas y utilidades necesarias para la aplicación.

- Capas de archivos que optimizan el almacenamiento y permiten la reutilización.

Cuando usas una imagen, puedes crear un **contenedor**, que es como una versión activa de esa imagen, ejecutándose en tu computadora o en la nube asegurando que todo funcione sin problemas en cualquier entorno. Este contenedor actúa como un entorno aislado donde la aplicación se ejecuta de manera uniforme, sin importar dónde se despliegue.

Cuando ejecutas una imagen, Docker crea un contenedor, que es una instancia en funcionamiento de esa imagen corriendo en tu ordenador o en la nube. Este contenedor funciona como un entorno aislado, asegurando que la aplicación se ejecute de forma consistente y sin problemas, sin importar el lugar en el que se despliegue.

Por ejemplo, imagina que tienes una aplicación web. La imagen Docker incluiría:

1. *El sistema operativo necesario* – Un sistema operativo liviano como `Alpine Linux` o `Ubuntu`, que proporciona la base mínima para ejecutar la aplicación.
2. *El servidor web* – Un software como `Nginx` o `Apache`, que se encarga de gestionar las peticiones de los usuarios y servir las páginas web.
3. *Tu aplicación con su código* – Puede ser una aplicación escrita en `Node.js`, `Python (Django/Flask)`, `PHP (Laravel)`, o cualquier otro lenguaje de programación que necesite para ejecutarse.
4. *Las herramientas necesarias, como* –
   - Un motor de base de datos como `MySQL` o `PostgreSQL` para almacenar datos de usuarios.
   - Librerías de programación, por ejemplo, `Express` para aplicaciones en Node.js o `Gunicorn` para aplicaciones en Python.
   - Dependencias de sistema como `curl` o `git` para descargar y administrar archivos.
5. *Las configuraciones necesarias, como* –
   - Variables de entorno para definir parámetros como el puerto de ejecución (`PORT=8080`), credenciales de la base de datos (`DB_USER=root`, `DB_PASS=secret`).
   - Archivos de configuración como `nginx.conf` para personalizar cómo el servidor web gestiona las solicitudes.
   - Configuraciones de seguridad, como certificados SSL para conexiones seguras (HTTPS).

Contenedor Docker

Un contenedor Docker es una instancia en ejecución de una imagen Docker.

En otras palabras, es el entorno aislado donde se ejecuta tu aplicación. Si tomamos el ejemplo de una aplicación web, el contenedor sería el espacio donde el servidor web de la aplicación está activo y puede responder a solicitudes de usuarios.

Características principales de un contenedor —

- **Aislamiento**: Los contenedores están aislados unos de otros y del sistema operativo host. Esto significa que cada contenedor puede tener su propio sistema de archivos, variables de entorno y procesos sin interferir con otros contenedores o el sistema operativo principal.
- **Ligereza**: Los contenedores son mucho más ligeros que las máquinas virtuales (VMs) porque no requieren un sistema operativo completo. Solo comparten el núcleo del sistema operativo anfitrión, lo que hace que los contenedores sean más rápidos y menos pesados.
- **Portabilidad**: Un contenedor se puede ejecutar en cualquier entorno que tenga Docker instalado. Esto incluye tu máquina local, servidores en la nube, o incluso en máquinas de desarrollo de otras personas, y siempre funcionará de la misma manera, ya que el contenedor contiene todo lo necesario para ejecutar la aplicación (código, dependencias, configuraciones).
- **Inmutabilidad**: Los contenedores son inmutables, lo que significa que una vez que un contenedor está creado, no puedes cambiar su estado directamente. Si necesitas hacer cambios (como actualizar el código o las dependencias), deberás crear una nueva imagen y luego ejecutar un nuevo contenedor a partir de ella.
- **Escalabilidad**: Puedes ejecutar múltiples instancias de un contenedor para escalar aplicaciones fácilmente. Docker y otras herramientas como Kubernetes permiten gestionar contenedores de manera eficiente y automatizada. Al escalar una aplicación por demanda se pueden añadir más contenedores que se comunicarán entre si repartiendose la demanda o distribuyendo las tareas. Por ejemplo, si una tienda online recibe miles de visitas durante eventos especiales como el Black Friday. Con la escalabilidad horizontal, se pueden desplegar más contenedores de la aplicación para manejar el aumento de pedidos sin que la web se vuelva lenta o colapse.
- **Interacción con el sistema host**: Los contenedores pueden acceder a ciertos recursos del sistema operativo anfitrión, como puertos de red, volúmenes de almacenamiento, o variables de entorno, pero sin tener acceso completo al sistema operativo, lo que aumenta la seguridad.

Ciclo de vida de un contenedor —

1. **Creación**:

   Cuando creas un contenedor, Docker toma una imagen y la convierte en una instancia en ejecución. Esta instancia está completamente aislada y tiene acceso a todos los recursos necesarios que se definieron en la imagen (por ejemplo, código, dependencias, variables de entorno).
2. **Ejecución**:

   Mientras está en ejecución, el contenedor ejecuta la aplicación definida por la imagen y expone los puertos y recursos necesarios para interactuar con él, como un servidor web o una base de datos.
3. **Detención**:

El contenedor puede ser detenido en cualquier momento. Cuando se detiene, todos los procesos dentro del contenedor se detienen, pero los cambios en los volúmenes persistentes (si los hay) no se pierden.

4. **Eliminación**:

   Cuando un contenedor se elimina, cualquier cambio realizado dentro del contenedor (si no se ha persistido) se pierde. Sin embargo, la imagen de la que se creó el contenedor sigue existiendo, y puedes crear nuevos contenedores a partir de esa imagen.

Los **volúmenes** son una forma de persistir datos fuera de los contenedores. Si un contenedor almacena datos solo en su propio sistema de archivos, esos datos se perderán cuando el contenedor sea eliminado. Los volúmenes, en cambio, son almacenamiento persistente que se puede compartir entre contenedores y persistir incluso si el contenedor se detiene o elimina.

Por ejemplo, si tienes un contenedor que necesita almacenar datos de una base de datos, puedes usar un volumen para que esos datos se mantengan incluso si el contenedor se elimina.

Para crear un contenedor con un volumen:

```
docker run -v /ruta/del/host:/ruta/del/contenedor <nombre_imagen>
```

Redes de contenedores —

Los contenedores pueden comunicarse entre sí a través de redes. Docker crea una red predeterminada, pero también puedes definir redes personalizadas para que los contenedores interactúen entre ellos de manera más controlada.

Por ejemplo, si tienes un contenedor para tu base de datos y otro para tu aplicación, puedes crear una red para que ambos puedan comunicarse entre sí.

```
docker network create mi_red
docker run --network mi_red <nombre_imagen>
```

- Comandos básicos para trabajar con contenedores docker
  - **Listar contenedores en ejecución**:

    ```
    docker p
    ```

  - **Listar todos los contenedores (incluyendo los detenidos)**:

    ```
    docker ps -a
    ```

- **Iniciar un contenedor/ crear y ejecutar un nuevo contenedor basado en una imagen**:

```
docker run <nombre_image
```

- **Detener un contenedor**:

```
docker stop <nombre_o_id_del_contenedor
```

- **Eliminar un contenedor**:

```
docker rm <nombre_o_id_del_contenedor>
```

- **Ejecutar un contenedor en segundo plano (detached)**:

```
docker run -d <nombre_imagen>
```

- **Acceder a la terminal de un contenedor en ejecución**:

```
docker exec -it <nombre_o_id_del_contenedor> /bin/bash
```

Los contenedores son más ligeros, rápidos y se ejecutan directamente sobre el núcleo del sistema operativo host. No requieren un sistema operativo completo, solo las librerías y dependencias necesarias para ejecutar la aplicación. Por eso consumen menos recursos y en muchos casos son más recomendables que las máquinas virtuales.

//estos los iria enseñando a modo de ejemplo practicando usando docker hub

> vams a hacer un ejemplo >>> abriendo windows powershell >>> vamos a usar PostrgreSQL (o cualquier imagen que encontremos en el Docker Hub, y de ahí ir enseñando los comandos) **EN ESTE EJEMPLO USAMOS UNA IMG YA HECHA**

Comandos más comunes en Docker

docker run → descarga y la ejecuta la imagen

docker pull → descarga la imagen pero no la ejecuta

docker images → mostrar las imagenes que están en nuestra maquina

docker ps → contenedores ejecutandose

docker ps -a → muestra los contenedores q se estan ejecutando y los anteriores tmb

docker start [container id]→ reiniciar un contenedor para que conserve con los datos q ya tenia (no se suelen guardar datos importantes ya q en cualquier momento los puedes destruir)

docker logs [container id] → ver la salida, resultados de cuando se ha ejecutado el contenedor, los errores o la salida estandar

docker logs [container id] -f → salida del contenedor en tiempo real de ejecucion. Como si fueran los console.log en javascript o los logs en la terminal de kotlin que usamos para ver si el emulador se ha arrancado correctamente o no . **-f ⇒ es de follow**

docker exec [container id]→ ejecuta un comando dentro de un contenedor q ya esta corriendo

- **-it** ⇒la **i** lo que va a hacer es crear una "sesion interactivo", y la **t** va a emular una terminal, como por ejemplo powershell, por lo que el comando completo quedaría ⇒ **docker exec -it [container id] sh**
  - contenidos del contenedor dentro de la terminal de powershell → ls

docker stop [container id] → detener el contenedor

docker run -d [container id] [nombre img q querems ejecutar]→ correr un contenedor en background (ej. docker run -d [id] postgres)

> 🔥 **termino importante ⇒ DAEMON, se ejecuta en segundo plano y maneja los contenedores …..**
>
> **que se encarga de:**
> **+ escuchar las solicitudes que hacemos los usuarios**
> **+ crear y ejecutar contenedores**
> **+ manejar el alamacenamiento**
>
> ⇒ si docker daemon no está corriendo, no podremos usar docker. Para comprobar que está activo **docker info** (en la terminal de powershell)

CONSTRUIMOS NUESTRA IMGEN

- qué nos facilita docker 🚨 ✅

  Primero de todo, que uso tiene, o que ventajas tiene la creacion y uso de las imagenes enDocker:

  ❌ Sin Docker

  En un proyecto por ejemplo de python, y alguien más quiere ejecutarlo, necesitaría tener la version de python correspondiente al proyecto, instalar las librerias necesarias para el

funcionamiento del proyecto, asegurarse del correcto funcionamiento…

✅ Con Docker

Solo necesitaríamos `**docker run [nombre de la img]**`

(en el ejemplo de helloworld, sería app-prueba o si no funciona hello-world-app)

Y si usaramos Docker compose, que facilita el uso de varias tecnologias al mismo tiempo, usaríamos el comando `**docker-compose up**` y arrancaría toda la estructura con este comando solamente

- **EJEMPLO1** **HELLO-WORLD**
  1. Nos creamos un archivo app.py → dnd vamos a hacer nuestro HelloWorld, pero usando python:

     `**print("Hello World desde docker!")**`
  2. Creamos nuestro Dockerfile ⇒ dnd especificaremos caracterisíticas como la versión de python que vamos a usar, el comando a usar para la ejecución del script…

```
\#DOCKERFILE

FROM pyhton:3.12

WORKDIR /app

\#copiamos el script qe hemos creado en la img vamos a crear
COPY app.py .

\#para ejecutarla por terminal
CMD ["app.py"]
```

  3. Desde la terminal de POWERSHELL, accedemos a la carpeta dnd tenemos nuestro archivo python y desde ahí ejecutaremos los siguientes comandos

```
docker build -t hello-python .
# -t --> para agregarle un tag, un nombre a nuestra img
# . --> le indicamos q el DOCKERFILE está en el directorio actual

docker run hello-python
# si quisieramos q al terminar el contenedor se destruyera --rm

docker ps # contenedores en ejecucion
docker rm [id del contenedor] \#esto por si al terminar quisieramos
eliminarlo
```

- **EJEMPLO2** - juego en python

//para este ejemplo voy a usar el codigo de python del juego q hice para el master

1. Nos creamos otro proyecto, con un archivo juego.py, que va a contener el siguiente script
   - codigo aqui 👇

```
import pygame  # para la creacion de nuestro juego
import random  # numeros random --> definiremos la posicion de las cartas de manera aleatoria
import os  # para acceder a la carpeta de las imagenes

# iniciamos Pygame
pygame.init()

# configuramos las dimensiones de la ventana
ancho, alto = 600, 600
ventana = pygame.display.set_mode((ancho, alto))

# colores q vamos a tener dentro de la ventana
WHITE, BLACK, BLUE, GRAY = (255, 255, 255), (0, 0, 0), (0, 0, 255), (169, 169, 169)
# fuente_juego = "PressStart2P-Regular.ttf"  PQQQQQQQQ NO FUNCIONASSSSS
FONT = pygame.font.Font(None, 36)

# Configuración del tablero
filas, columnas = (
    4,
    4,
)  # Definir las filas y columnas del tablero donde se van a encontrar las cartas
tamanio_cartas = ancho // columnas

# Cargar imágenes de cartas desde la carpeta 'assets'
assets_carpeta = os.path.join(os.path.dirname(__file__), "assets")
cartas = [
    pygame.image.load(os.path.join(assets_carpeta, f"{i}.jpg"))
for i in range(1, 9)
]


# adaptamos el tamaño de las cartas para q se ajustes al tablero
def cambiar_cartas_size(cartas, tamanio):
    return [pygame.transform.scale(carta, (tamanio, tamanio))
for carta in cartas]
```

```python
# Crear cartas
def crear_cartas(cartas, tamanio_cartas, filas, columnas):
    redimensionadas = cambiar_cartas_size(cartas,
tamanio_cartas)
    pareja_cartas = (
        redimensionadas * 2
    )  # duplicamos cada una de las cartas para podamos
encontrar las parejas en el juego
    random.shuffle(
        pareja_cartas
    )  # cada vez q el juego se este inicializando --> las
cartas serán mezcladas aleatoriamente --> las duplicadas tmb :v

    tablero_cartas = []  # nos creamos el tablero vacio -->
inicio del juego
    # dividimos las cartas en filas --> tantas FILAS como haya
    for i in range(filas):
        inicio = i * columnas
        fin = inicio + columnas
        fila = pareja_cartas[inicio:fin]
        tablero_cartas.append(fila)

    return tablero_cartas


# ventana de BIENVENIDA :)
def ventana_inicial():
    ventana.fill(WHITE)
    title_text = FONT.render("Juego de Memoria", True, BLACK)
    rules_text1 = FONT.render("Encuentra las parejas de
cartas.", True, BLACK)
    rules_text2 = FONT.render("Haz clic en una carta para
revelarla.", True, BLACK)
    rules_text3 = FONT.render("Si coinciden, se quedan
abiertas.", True, BLACK)
    rules_text4 = FONT.render("Si no, se volverán a ocultar.",
True, BLACK)

    ventana.blit(title_text, (ancho // 4, alto // 4))
    ventana.blit(rules_text1, (ancho // 4, alto // 2))
    ventana.blit(rules_text2, (ancho // 4, alto // 2 + 30))
    ventana.blit(rules_text3, (ancho // 4, alto // 2 + 60))
    ventana.blit(rules_text4, (ancho // 4, alto // 2 + 90))
    pygame.display.flip()


# creacion del tablero
def tablero():
    ventana.fill(WHITE)
    for row in range(filas):
```

```
        for col in range(columnas):
            x, y = col * tamanio_cartas, row * tamanio_cartas
            if cartas_reveladas[row][col]:
                ventana.blit(board[row][col], (x, y))
            else:
                pygame.draw.rect(
                    ventana, GRAY, (x, y, tamanio_cartas,
tamanio_cartas), 3
                )  # Añade borde alrededor de cada carta no
revelada
    pygame.display.flip()


def comprobar_pareja_cartas(selected):
    if len(selected) == 2:
        r1, c1 = selected[0]
        r2, c2 = selected[1]
        if board[r1][c1] == board[r2][c2]:
            return True
        else:
            pygame.time.wait(1000)
            cartas_reveladas[r1][c1] = True \#false
            cartas_reveladas[r2][c2] = True \#false
    return False


def victoria():
    # devolveremos true si hemos revelado todas las cartas -->
q será cuando hayamos ganado el juego
    if all(all(row) for row in cartas_reveladas):
        return True


def mostrar_mensaje_victoria():
    ventana.fill(WHITE)
    texto_victoria1 = FONT.render("¡Has ganado! :)", True,
BLACK)
    texto_victoria2 = FONT.render("¡Nos vemos en la proxima!",
True, BLACK)
    texto_victoria3 = FONT.render(f"Pulsaciones totales:
{pulsaciones}", True, BLACK)
    ventana.blit(texto_victoria1, (ancho // 2, alto // 3))
    ventana.blit(texto_victoria2, (ancho // 3, alto // 2))
    ventana.blit(texto_victoria3, (10,10))
    pygame.display.flip()
    pygame.time.wait(3000)


board = crear_cartas(cartas, tamanio_cartas, filas, columnas)
cartas_reveladas = [
```

```
    [False] * columnas for _ in range(filas)
]  # todas las cartas inicialmente ocultas --- inicio

# bucle principal de ejecución en el programa
en_ejecucion = True
selected = []
mostrar_titulo_ventana = True  # Controla si se muestra la
pantalla inicial
pulsaciones = 0 \#contador de pulsaciones

while en_ejecucion:
    if mostrar_titulo_ventana:
        ventana_inicial()
        for event in pygame.event.get():
            # con le objeto event cogemos las acciones o
pulsaciones con las que se pueden encontrar los usuarios

            if event.type == pygame.QUIT:  # 1º accion de
salida del juego
                en_ejecucion = False
            if (
                event.type == pygame.MOUSEBUTTONDOWN
            ):  # 2º event del usuario --> btn pulsado
                mostrar_titulo_ventana = False
    else:
        ventana.fill(WHITE)
        tablero()
        \#mostramos las pulsaciones y q se vayan actualizando
cada vez q pulsamos
        texto_pulsaciones = FONT.render(f"Pulsaciones:
{pulsaciones}", True, BLACK)
        ventana.blit(texto_pulsaciones, (10, 10))

        pygame.display.flip()\#esto es lo que nos permite la
actualizacion de la variable cada vez q el usuario selecciona
una carta

        for event in pygame.event.get():
            # volvemos a coger los eventos segun las
condiciones :)

            if event.type == pygame.QUIT:  # 1º salimos del
juego
                en_ejecucion = False

            #\#//SI DETECTAMPS Q EL RATON HA SIDO PULSADO -->
ejecutaremos el codigo
            if (
                event.type == pygame.MOUSEBUTTONDOWN
            ):  # 2º cogemos la pulsacion con el raton del
```

```
usuario
                pulsaciones += 1
                x, y = event.pos
                # 3º pillamos la posicion de dnd se encuentra
el ratón  --> q está siendo manejado por el usuario --> con las
posiciones "X" e "Y"

                #\#----------------------
                \#calculamos la posición de la carta en filas y
columnas
                fila = y // tamanio_cartas  \#con esto sabemos
en q FILA hemos echo el click
                columna = x // tamanio_cartas  \#y aqui en la
COLUMNA

                \#comprobacion de la carta  --> está oculta
                if not cartas_reveladas[fila][columna]:
                    # Si está oculta, revelarla
                    cartas_reveladas[fila][columna] = True

                    \#añadimos la carta seleccionada a la lista
de seleccionadas
                    selected.append((fila, columna))

                    \#Comprobamos si hay dos cartas
seleccionadas
                    if len(selected) == 2:
                        # revisamos si son iguales
                        son_pareja =
comprobar_pareja_cartas(selected)
                        if not son_pareja:
                            # Si no son iguales, ocultarlas de
nuevo
                            fila1, columna1 = selected[0]  #
Primera carta seleccionada
                            fila2, columna2 = selected[1]  #
Segunda carta seleccionada

                            cartas_reveladas[fila1][columna1] =
False  \#ocultamos carta 1
                            cartas_reveladas[fila2][columna2] =
False  \#ocultamos carta 2

                        \#vaciamos nuestro array de cartas
seleccionadas para dar paso al  siguiente turno :)
                        selected = []

                if victoria():
                    mostrar_mensaje_victoria()
                    en_ejecucion = False
```

```
                    # la variable de q el juego este en marcha
la ponemos en FALSE para que deje de ejecutarse y se cierre
automaticamente
pygame.quit()
```

2. Creamos el DockerFile (*no tiene extension, solamente es necesario llamarlo así)

```
FROM python:3.12
WORKDIR /app

\#copiamos todos los archivos del proyecto al contendor q estamos
creando
COPY . /app
\#copiamos las carpeta q contiene las imgs del proyecto
COPY assets /app/assets

RUN pip install pygame random2

CMD ["juego.py"]
```

3. Construir y ejecutar el contenedor

```
docker build -t container-juego .
docker run -it container-juego

/*este comando nos dará paso a que ejecutemos el comando
python juego.py --> establecido en el dockerfile */
```

🚨 ⚠️ ESTAR DENTRO DEL DIRECTORIO DND ESTAMOS TRABAJANDO

🚨 ⚠️ si hacemos cualquier cambio tenemos que RECONSTRUIR LA IMG → ya q funciona como una foto, no se modifica

- **EJEMPLO3** - cambiar archivos de directorio

//el ejemplo que vamos a realizar consiste en pasar archivos(imgs) de una carpeta a otra, de esta manera trabajar con rutas absolutas y ver como se pasarían ambos valores 🙂

1. En nuestro espacio de trabajo, vamos a tener dos carpetas, archivos_actuales y archivos_tratados
   Para q este proceso funcione, necesitamos un script que trabaje con esos archivos, y cree la carpeta de archivos_tratados, en caso de q el usuario cuando ejecute el codigo en su S.O nos aseguremos de q la tiene creada
   - codigo mover-archivos.py

```python
import os
import shutil

def mover_archivos():
    # Rutas relativas de las carpetas
    carpeta_actual = os.path.dirname(os.path.abspath(__file__))
    carpeta_nuevos = os.path.join(carpeta_actual,
"archivos_actuales")
    carpeta_tratados = os.path.join(carpeta_actual,
"archivos_tratados")

    # Crear carpeta de archivos tratados si no existe
    os.makedirs(carpeta_tratados, exist_ok=True)

    # Listar archivos en archivos_actuales
    archivos = os.listdir(carpeta_nuevos)

    if not archivos:
        print("No se han encontrado archivos en la carpeta
'archivos_actuales' para mover.")
        return

    print(f"Archivos encontrados en 'archivos_actuales':
{archivos}")
    print("Procesando archivos...")

    # Mover archivos de archivos_actuales a archivos_tratados
    for archivo in archivos:
        ruta_origen = os.path.join(carpeta_nuevos, archivo)
        ruta_destino = os.path.join(carpeta_tratados, archivo)

        if os.path.isfile(ruta_origen):  # Verifica que sea un
archivo
            shutil.move(ruta_origen, ruta_destino)
            print(f"Archivo '{archivo}' movido a
'{carpeta_tratados}'.")

    print("Todos los archivos han sido procesados.")

if __name__ == "__main__":
    mover_archivos()
```

2. Creamos el Dockerfile:

```dockerfile
FROM pyhton:3.12

# directorio de trabajo dentro del contenedor
WORKDIR /app
```

```
COPY mover-archivos.py /app

CMD ["python", "mover-archivos.py"]
```

3. Creamos la img desde la terminal de powershell
   👀 ojo! ⇒ estar en el directorio dnd estamos trabajando

```
\#creamos la img con un tag (le asignamos nombre nuestra img)
docker build -t img-mover-archivos
```

   - funcionalidad adicional, para **compartir** y **cargar una imagen** `**docker save -o compartir-img.tar img-mover-archivos**` **-o** → para omitir informacion del usuario q crea y q envia la imagen **compartir-img.tar** → nombre de la carpeta dnd metemos la img **img-mover-archivos** → la img que metemos en la carpeta .tar
     - CARGAMOS LA IMG.TAR `**docker load -i compartir-img-tar**`
4. Ejecutamos la imagen

```
docker run -v “ruta absoluta de la carpeta de
inicio(archivos_actuales:/app/archivos_actuales)”
-v “ruta de la carpeta de destino
(archivos_tratados:/app/archivos_tratados)”
[nombre de la img]
```

5. visualizar la img desde la terminal → desde su nucleo (la carpeta /app)

   docker run -it --entrypoint sh [nombre de la img]
   ⇒ cambia la vista de la terminal y ahora empieza con #

```
# pwd
/app
# ls
mover-archivos.py
# cat mover-archivos.py --> con esto visualizamos el contenido
codigo python de nuestra img
```

- cositas a tener en cuenta :) asi evitamos traumitas

> 🔥 **SI QUIERES USAR REPETIDAS VECES EL CONTENEDOR, NO USES ESTE COMANDO jajaja**
> **docker run --rm --it [container id] sh ⇒ si ejecutamos este comando luego no funciona y el daemon de docker te dice q no lo encuentra, PQ RM → ES**

> **DE REMOVEEE, SE CREA EL CONTENEDOR Y CUANDO ACABAMOS DE USARLA SE DESTRUYE 🙂**

Ampliar temas

- Docker Compose
- multi stage building
- **Pautas**