INTRODUCCIÓN (CONTENEDORES E IMÁGENES)

- **¿Qué es Docker?** Software de **virtualización** que **facilita el desarrollo y el despliegue de aplicaciones**.

- **¿Cómo funciona?** Ejecuta una aplicación dentro de un "**contenedor**" (container) junto con todas las dependencias, configuraciones y herramientas de sistema y de entorno necesarias. Es decir, **empaqueta la aplicación junto con todo lo que necesita para funcionar**. Estas imágenes o contenedores son **portables**; fáciles de distribuir y desplegar.

- **¿Porque facilita el trabajo?** Los contenedores ofrecen un **entorno aislado**, ya preconfigurado con todo lo necesario para trabajar con la aplicación a desarrollar. Es decir, un contenedor es **independiente del sistema operativo y de los servicios de la máquina donde se está ejecutando**. En caso de que se requiera un nuevo servicio, se arranca como otro contenedor que trabaje en conjunto, o se añade al empaquetado. Con esto, se logra **estandarizar el proceso de preparar los servicios** necesarios de forma local, **permitiendo centrarse en el desarrollo del programa**.

- **¿Porque facilita el despliegue?** Los contenedores ofrecen abstracción sobre las dependencias y configuraciones necesarias para el código a ejecutar en el servidor, dado que ya viene todo preparado: **no se requieren dependencias o configurar específicas en el servidor por cada aplicación**.

- **Docker vs VirtualBox (u otras máquinas virtuales)** Si simplificamos el funcionamiento de un Sistema Operativo, podemos separarlo en dos capas principales:
  - ➔ Capa de **aplicación** - donde se ejecutan las aplicaciones (software)
  - ➔ Kernel (**core**) - interactúa entre los componentes hardware y software. Por debajo del Kernel tendremos todo el **hardware** del equipo.

  Las **máquinas virtuales** convencionales (como las que se montan con VirtualBox) **virtualiza el SO completamente; la capa de aplicación Y Kernel**. para negociar con el SO Hosts, las máquinas virtuales incluyen una capa intermedia denominada "Hypervisor".

  Docker virtualiza **únicamente la capa de aplicación**. En esta capa vendrá los servicios y las aplicaciones indicadas, todo ya instalado en el contenedor. Para que esta capa de aplicación virtualizada sea compatible con todos los SO, Docker también añade una capa de "Hypervisor", que consiste en una distribución de linux ligera (parte del motor/daemon).

  Al comparar ambas virtualizaciones tendremos que:
  - **El tamaño** de los contenedores o imágenes **de Docker es mucho menor**. Una imagen de Docker puede ocupar algunos MBs, mientras que las máquinas virtuales ocupan GBs.
  - **Arrancar contenedores de Docker es mucho más rápido** que arrancar máquinas virtuales.
  - Técnicamente, **Docker solo es compatible con distribuciones de linux**, mientras que las máquinas virtuales son compatibles con todos los SOs.

  El problema de la compatibilidad se produce porque los contenedores de Docker se basan en linux. Originalmente, Docker fue desarrollado para este SO. Con **Docker Desktop** se añade la capa de "Hypervisor", que consiste en una distribución de linux ligera. Esto permite que se ejecuten contenedores en windows o MacOS sin problemas.

- **Docker Desktop**. La instalación de Docker Desktop incluye:

  - ➢ **Motor Docker** - hace posible la virtualización de contenedores
    - Se trata de un servidor con un demonio llamado "dockerd"
  - ➢ **Cliente CLI** Docker - Command Line Interface "docker"
    - Permite interactuar con el servidor de Docker vía comandos
  - ➢ **Cliente GUI** - Graphical User Interface
    - Permite interactuar con el servidor de Docker vía ventana/app (user friendly)
  - ➢ **Docker Build** - Para crear contenedores; herramienta de empaquetado
  - ➢ **Docker Compose** - Para configurar contenedores con múltiples aplicaciones
  - ➢ Docker Content Trust - Para verificar el origen de una imagen de Docker
  - ➢ **Kubernetes** - Automatizar despliegue de aplicaciones
  - ➢ Credential Helper - Suite para mantener credenciales de docker seguros.
  - ➢ Extensiones

- **Images vs Containers**

  La imagen de Docker es el conjunto o paquete de la app al completo:
- Artefacto de aplicación ejecutado (como un .jar o .war)
- No solo tiene la aplicación a ejecutar; también incluye:
  - Opcionalmente; el código de la app (.java)
  - Entorno de desarrollo completamente configurado (Linux, node.js, npm…)
  - Variables de entorno, directorios necesarios, etc (estructura de carpetas)

  El **contenedor** de Docker **es una instancia en ejecución de una imagen** de Docker. Es el programa desarrollado en funcionamiento. Cuando se inicia un contenedor es cuando se crea todo el entorno definido en la imagen. En esencia; **un contenedor es una instancia de una imagen**. Es posible ejecutar múltiples instancias de la misma imagen (múltiples contenedores).

- **Registros de Docker (Docker Registries)**. Un registro de Docker es un **almacén en cloud** para la distribución de imágenes de Docker. ¡Ver Docker Hub; el almacén oficial de Docker! Ofrece imágenes oficiales de aplicaciones.

- **Registry vs Repository**
  - Docker Registry es un servicio que provee almacenamiento para imágenes.
  - Puede ser hospedado por terceros o por uno mismo.
  - Dentro de docker registry es posible tener múltiples repositorios.

En conclusión, dentro de un Registro de Docker, cada aplicación tendrá su propio repositorio, y dentro de dicho repositorio se almacenarán diferentes versiones de imágenes de esa aplicación.

DOCKER - FUNCIONAMIENTO (GESTIÓN DE IMÁGENES Y CONTENEDORES)

- **Pull.** Entendemos por Pull **descargar una imagen** de docker hub (registros de imágenes) y almacena en local. Vía terminal, la estructura del comando Pull es: ***docker pull [Nombre_imagen]:[Etiqueta_version]***

  Ejemplo de Pull para preparar un servidor web **nginx** (similar a apache) versión : *docker pull nginx:1.27*

  Por defecto el Pull se realiza sobre docker hub (docker.io). Es posible usar el tag "latest", o no indicar tag, para descargar la última versión, pero siempre es recomendable emplear un tag de versión específico (buena práctica). Para **ver las imágenes descargadas** y sus tags (etiquetas), emplear el comando; ***docker images***

- **Run.** Mediante el comando Run **se crea y se arranca un contenedor** con la imagen que se le especifique: ***docker run [Nombre_imagen]:[Etiqueta]***
  En el caso de la imagen de nginx descargada: *docker run nginx:1.27.* Al ejecutar el comando, por consola se mostraran logs de nginx inicializando y en funcionamiento. El terminal queda reservado a nginx.

- **ps.** El comando "ps" representa "**process status**" y sirve para ver el estado de los contenedores en marcha: **docker ps**
  Muestra el ID de contenedor, la imagen sobre la que se ha arrancado, cuando fue creado, su estado, puertos en uso y su "nombre". Docker genera nombres aleatorios para los contenedores automáticamente si no se especifica uno.

- **Run -d.** Mediante el comando Run -d se crea y se arranca un contenedor con la imagen que se le especifique, pero en **modo "detach"** para no bloquear el terminal: ***docker run -d [Nombre_imagen]:[Etiqueta]***

  En el caso de la imagen de nginx descargada: *docker run -d nginx:1.27.* Al ejecutar el comando, por consola se mostrará el ID completo del contenedor, pero sin bloqueo.

- **logs.** Con un contenedor arrancado en modo detach ya no es posible ver los logs con consola. En caso de que queramos verlos, es posible mostrarlos vía comando: ***docker logs [ID_Contenedor]***

  Nota: Basta con la ID de contenedor corta (primeros caracteres, ver docker ps). Esto nos muestra por pantalla los registros generados por el contenedor hasta ese momento, sin bloquear la terminal. También es posible emplear el **nombre de contenedor**.

- **stop.** Para detener uno o más contenedores en marcha: ***docker stop [ID_Container]***

  Nota: Basta con la ID de contenedor corto (primeros caracteres, ver docker ps). Se pueden poner sucesivos IDs para detener varios contenedores simultáneamente. También es posible emplear el **nombre de contenedor**.

- **Sobre Pull & Run.** No es necesario realizar manualmente la descarga de imagen mediante Pull:
  Run realiza un Pull de la imagen sobre docker hub si no la localiza en local.
  Si queremos arrancar otra versión anterior de nginx, la 1.23, que no está descargada, basta con hacer: *docker run nginx:1.23*

- **Container Port vs Host Port.** La aplicación en funcionamiento dentro de un contenedor se encuentra en una red de docker **aislada**, separada de la red del equipo local. Esto **permite poder tener varias aplicaciones en funcionamiento trabajando sobre el mismo puerto**. Para poder acceder a la aplicación, se debe **exponer el puerto que usa el contenedor al puerto del host** (la máquina local sobre la que se ejecuta el contenedor). Para ello se deben enlazar los puertos (**port binding**).

- **Port Binding.** Se puede emplear **cualquier puerto del host** al enlazar con los puertos de un contenedor. En el ejemplo de Nginx, igual que apache, funciona sobre el puerto 80 por defecto. Se puede asignar el puerto del host que se desee (ej. 9000) para vincularlo con el puerto 80 del contenedor. Aun así, el estándar consiste en enlazar el mismo puerto local que se requiere en el contenedor (buena práctica. ej, 80:80).

- **Port Binding -run -p.** El enlace de puertos se debe configurar al arrancar un contenedor. Para esto, al comando **Run** se añade el flag "**-p**" o "**--publish**" para publicar el puerto del contenedor al host.
  docker run ***-p [Puerto_Host]:[Puerto_contenedor]*** *[Nombre_Imagen]:[Etiqueta_ver]*

  Para hacer accesible un contenedor de Nginx mediante el puerto 9000: *docker run -d -p 9000:80 nginx:1.27*

  Este comando permite acceder a la página por defecto de Nginx vía: localhost:9000. Tras acceder, si se visualizan los logs del contenedor se pueden observar peticiones GET. Solo se puede definir **un puerto del host por servicio**. El comando ps muestra este enlace de puertos.

- **ps -a.** Cada vez que se ejecuta un "docker run" se está creando un nuevo contenedor, **NO se están reutilizando**. docker ps solo nos muestra los contenedores en funcionamiento. Sin embargo, si añadimos el flag "**-a**" o "**--all**" se **mostrará la lista completa de contenedores**, tanto en funcionamiento como parados. *docker ps* ***-a*** **start**

  Para reutilizar un contenedor ya creado previamente, se puede emplear el comando start: ***docker start [ID_Contenedor]*** Nota: Basta con la ID de contenedor corto (primeros caracteres, ver docker ps). Se pueden poner sucesivos IDs para iniciar varios contenedores simultáneamente. También es posible emplear el **nombre de contenedor**.

- **Nombrar un contenedor - run --name**. Al crear un contenedor, mediante el flag "**--name**" es posible **otorgar un nombre específico** (no

aleatorio) al contenedor que se vaya a crear. Esto facilita mucho su gestión (por ejemplo, para arrancar y detener por consola, o ver logs).

*docker run **--name [Nombre_contenedor]** [Nombre_Imagen]:[Etiqueta_ver]*

Aplicado sobre el ejemplo de Nginx queda: *docker run --name web-server -d -p 9000:80 nginx:1.27*

DOCKER - BUILD (DOCKERIZE)

- **Desplegar una aplicación sobre un servidor**. Con Docker, una vez desarrollada una aplicación, se debe desplegar en el servidor dentro de un contenedor. Para esto, el programa y todos los recursos necesarios para que funcione se deben empaquetar en una imagen. En el servidor se pueden desplegar más contenedores necesarios para el funcionamiento de la aplicación (pe. Base de datos mysql).

- **Crear una imagen de aplicación.** Se debe definir cómo se ha de construir la imagen: para ello se genera un fichero "Dockerfile" que contiene las instrucciones de montaje de la imagen.

- **Ejemplo de aplicación node.js**
  tutorial/src/server.js:
  const express = require('express');
  const app = express();
  app.get('/', (req,res)=>{ res.send("¡Hola Mundo!"); });
  app.listen(3000, function () { console.log("escuchando puerto 3000"); });

  tutorial/package.json:
  ```
  {
      "name": "mi-app",
      "version": "1.0",
      "dependencies": {
          "express": "4.18.2"
      }
  }
  ```
- **Ejemplo dockerfile para aplicación node.js**

  tutorial/Dockerfile:
  ```
  FROM node:19-alpine
  COPY package.json /app/
  COPY src /app/
  WORKDIR /app
  RUN npm install
  CMD ["node", "server.js"]
  ```

- **Estructura de Dockerfile - FROM**. Los Dockerfiles comienzan haciendo uso de una "**imagen base**". Esta imagen contiene, como mínimo, una **versión de linux ligera** sobre la que todo se va a ejecutar.
  Este SO es el que tendrá instaladas las herramientas que se vaya a emplear. La elección de esta imagen base dependerá de dichas herramientas. Por tanto, los **Dockerfiles deben comenzar con una instrucción FROM**. El concepto de apilar imagen sobre imagen significa que se irán formando múltiples capas. Este aporta el beneficio de poder tener imágenes en caché.

- **Estructura de Dockerfile - RUN**. Mediante RUN en Dockerfile se indican comandos a ejecutar al construir el contenedor. La directiva FROM declarada en el ejemplo anterior incluye npm, pero se requiere instalar en el SO, por tanto, se debe indicar la instrucción: *RUN npm install*
- **Estructura de Dockerfile - COPY**

  Copia directorios o ficheros y los añade al sistema de ficheros del contenedor, dentro de la ruta "**dest**".
  A diferencia de RUN que ejecuta comandos dentro del contenedor, **COPY se ejecuta en la máquina HOST**. Se puede indicar una ruta específica donde copiar. Si esta ruta termina en /, Docker interpreta que debe crear las carpetas en el contenedor si no existen.
  COPY package.json /app/
  COPY src /app/

- **Estructura de Dockerfile - WORKDIR.** WORKDIR sirve para indicar el directorio de trabajo. Específica donde se van a ejecutar los comandos que se declaren a continuación. Equivale a cambiar de directorio ("cd").
  WORKDIR /app

- **Estructura de Dockerfile - CMD**. La instrucción CMD indica el comando que se debe ejecutar cuando se arranque el contenedor. **Solo puede haber una instrucción CMD en un Dockerfile**, y, por tanto, normalmente será siempre la última a declarar.

  Se le debe pasar, en forma de parámetros de un array, el comando y su valor. *CMD ["node", "server.js"]*

- **Construir la imagen - Build.** Una vez creado el fichero Dockerfile, se puede construir la imagen mediante el comando **build** de Docker, **indicando la ruta** donde se encuentra el fichero: docker **build [Ruta_de_Dockerfile]**

  El flag **-t** o **--tag** sirve para darle un **nombre** a la imagen **y** (opcionalmente) una **etiqueta de versión**, empleando el formato "nombre:tag". Para generar la imagen de "mi-app" se puede emplear: *docker build -t mi-app:1.0* .

El punto (.) indica que se utilice la ruta desde donde se ejecuta el comando.

- **Estructura de Dockerfile y Construir la imagen**. La ejecución de Dockerfile es *secuencial*; **cada instrucción crea una capa o estado dentro de la imagen**. Esto se puede ver claramente en los logs de la construcción de la imagen, donde cada instrucción declarada es un paso con sus propios registros. *Recordatorio*: usar "docker images" para ver las imágenes descargadas o creadas.

- **Run de mi-app.** Ahora que se ha generado la imagen, es posible crear el contenedor y ejecutar el programa mediante el comando RUN: ***docker run -d -p 3000:3000 mi-app:1.0***
  Ahora **localhost:3000** debería mostrar "¡Hola Mundo!" vía navegador.

  Ver también los **docker logs** para este contenedor; debería mostrar "escuchando puerto 3000".

DOCKER - DESARROLLO WEB CON DOCKER (DOCKER NETWORK Y COMPOSER DE CONTENEDORES (YAML))

- **Docker Network.** Docker establece una red aislada del host, solo para contenedores. Para conectar contenedores en la red de Docker no se requiere asignar puertos; basta indicar los IDs o nombres de los contenedores.
  Para ver las redes que crea Docker: ***Docker network ls***

  Crear una nueva red de Docker: ***Docker network create [nombre_red]***

  Al arrancar (run) contenedores se debe incluir ***--net [nombre_red]*** para añadirlo a la red de Docker creada.

  Ejemplo:
  Docker network create mongo-network

  Nota: Recomendación para próximos ejemplos:
  docker pull mongo-express:0.49
  docker pull mongo:4.2.1

- **Variables de entorno de imagenes - run -e**. Algunas imágenes pueden incluir variables de entorno configurables. Por ejemplo, para sobreescribir usuario y contraseña por defecto en una base de datos. Estas variables se indicarán en su documentación y se pueden gestionar con el flag **-e** al arrancar el contenedor con la imagen (run).

  Ejemplo para MongoDB:
  docker run -p 27017:27017 -d
  -e MONGO_INITDB_ROOT_USERNAME=admin
  -e MONGO_INITDB_ROOT_PASSWORD=password
  -e MONGO_INITDB_DATABASE=user-account
  --name mongo --net mongo-network
  mongo

- **Docker Network**. Para añadir contenedores a la red, al arrancar con run les indicamos la misma red.

  Ejemplo: añadir Mongo Express para poder gestionar la base de datos MongoDB de la red Docker (PhpMyAdmin para MongoDB):

  docker run -d -p 8081:8081
  -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin
  -e ME_CONFIG_MONGODB_ADMINPASSWORD=password
  -e ME_CONFIG_MONGODB_SERVER=mongo
  --name mongo-express
  --net mongo-network
  mongo-express

  Para acceder a Mongo Express en localhost:8081 usar admin / pass.
  Acceder y crear una colección (collection) llamada "users" (tabla).

- **Docker Compose.** Como hemos visto, arrancar los contenedores y posteriormente gestionarlos vía comandos puede resultar una tarea tediosa. Docker Compose nos ayuda a facilitar esta tarea definiendo instrucciones sobre **cómo deben arrancar múltiples contenedores** y **trabajar** entre ellos.

- **Docker Compose - Estructura**
  Aspectos clave:
  - Fichero con extensión .yaml o .yml.
  - La indentación se debe respetar!
  - Comienza por "version" siempre; indica la versión de composer a usar.
  - Dentro de services se indica el nombre de los contenedores a arrancar.
  - Dentro de cada contenedor se define la imagen a usar y las propiedades del contenedor (puertos, variables de entorno, etc).

- **Docker Compose - Arrancar contenedores**. Mediante el comando docker-compose se puede arrancar los contenedores indicados en un fichero .yml o .yaml, indicando que realice un "up". ***docker-compose -f [fichero.yml] up***

  Ejemplo: *docker-compose -f mongo-docker-compose.yaml up*

  Al revisar los logs resultantes se puede ver el nombre de la red Docker que se ha creado. Si accedemos a localhost:8081 veremos que no existe la BD user-accounts y la colección users creada previamente.

- **Docker Compose - Detener contenedores.** Con docker-compose, indicando el fichero .yaml o .yml con los contenedores a detener, pero esta vez indicando "down". *docker-compose -f [fichero.yml]* ***down***

  Ejemplo: *docker-compose -f mongo-docker-compose.yaml down*

  Al detener y volver a arrancar el mismo fichero .yaml, docker compose reactiva los contenedores y su red asociada.

- **Docker Volumes**
  Mediante Docker Volumes se puede lograr tener datos persistentes en contenedores. Esto se logra asociando un fichero virtual del contenedor a un fichero real del Host. Esto implica guardar datos de contenedores de forma local.

  ➔ **Tipos**
  Formas de gestionar Docker volumes:
  - **Host volume**: *docker run -v [directorio_fisico:directorio_virtual]* (referencia directa de carpetas)
  - **Anonymous volume**: *docker run -v [directorio_virtual]* (Docker se encarga de guardarlo en el Host)
  - **Named volume**: *docker run -v [nombre_carpeta:directorio_virtual]* (Permite referenciar el volumen por el nombre dado, sin necesidad de saber la ruta)

  Normalmente se almacenan en .\docker\volumes del Host.ç

  ➔ **Docker Compose**

```
version: '3'
services:
    mongo:
        image: mongo
        ports:
            - 27017:27017
        environment:
            - MONGO_INITDB_ROOT_USERNAME=admin
            - MONGO_INITDB_ROOT_PASSWORD=password
        volumes:
            - db-data:/data/db          # define un named volume y la ruta virtual a guardar (¡ver docs!)

(...)

# al final del fichero se debe listar las unidades declaradas
volumes:
    db-data:
        driver: local          # Almacenamiento local en el Host
```

PRÁCTICA 10

Para poder subir estas imágenes, se deben empaquetar como .tar, esto se hace con el comando:
docker save [ID_o_Nombre_imagen]

Estas imágenes se pueden desempaquetar con el comando:
docker load [archivo.tar]

Tambien es muy recomendable, para evitar problemas, crear las imágenes de tu aplicación INDICANDO DIFERENTES VERSIONES para cada parte de la practica.

Por ejemplo;
para la parte 1: Docker Build -t mi_app_web:1.1 .

para la parte 2: Docker Build -t mi_app_web:1.2 .

para la parte 3: Docker Build -t mi_app_web:1.3 .