En este taller vamos a cubrir los aspectos mas importantes que se han utilizado en el proyecto IdPnube. Empezaremos conociendo Docker y a crear imágenes seguras. Para el proyecto en si, debido a la necesidad de automatizar la gestión y despliegue de contenedores en entornos complejos. Tener varias aplicaciones distribuidas en múltiples contenedores y necesitar administrar recursos, balancear la carga, asegurar la alta disponibilidad y gestionar fallos, incorporamos un orquestador para facilitar estas tareas, coordinando el funcionamiento de cada contenedor y manteniendo el entorno estable y eficiente.

Como ya sabréis Docker dispone de un orquestador, Docker swarm pero en este caso vamos a hablar de kuberentes, que es el que usamos por diversas funcionalidades:

* Estabilidad y robustez. Mayor numero de contenedores de forma mas eficiente
* Alta disponibilidad y autoreparacion. Mantiene los contenedores arriba de forma constante
* Control sobre despliegues y rollbacks
* Seguridad avanzada. Mayor control de acceso y autenticación

Docker

Docker es una herramienta clave en el presente y futuro de la tecnología. Desde aplicaciones locales hasta despliegues en la nube, Docker y los contenedores se encuentran en el centro de la mayoría de las arquitecturas modernas, como Kubernetes (k8s), y su rol sigue creciendo con innovaciones emergentes como WebAssembly (WASM).

Para comenzar, entendamos qué es un contenedor. Los contenedores son entornos aislados y ligeros, similares a máquinas virtuales, pero mucho más eficientes en el uso de recursos. Docker facilita la creación, ejecución y gestión de estos contenedores, proporcionando una solución versátil para desarrolladores y equipos DevOps.

Todas las aplicaciones que ejecuta Docker, las denominamos aplicaciones contenerizadas

**Tipos de Docker: Community y Enterprise Edition**

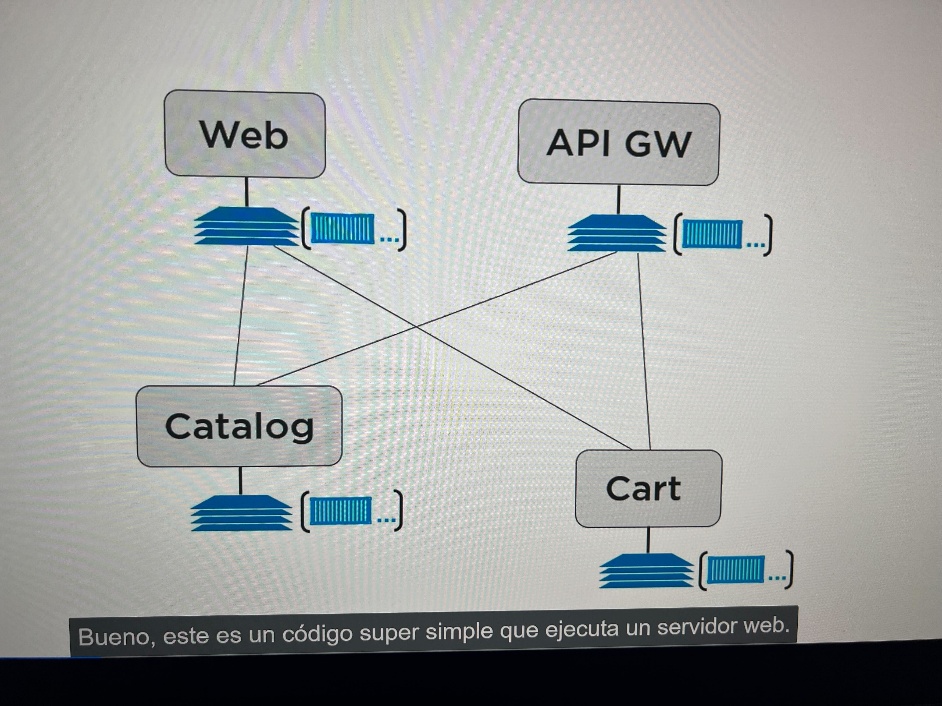
* **Community Edition (CE)**: Docker CE es gratuito, open-source y cuenta con lanzamientos frecuentes que integran rápidamente nuevas funcionalidades. Los usuarios pueden contribuir a su desarrollo y recibir actualizaciones periódicas.
* **Enterprise Edition (EE)**: Docker EE está diseñado para entornos empresariales, ofreciendo un soporte oficial y lanzamientos menos frecuentes, lo cual asegura mayor estabilidad en entornos de producción.

**Ventajas de Docker y los Contenedores**

1. **Portabilidad y Consistencia**
   * Docker permite empaquetar aplicaciones con todas sus dependencias, asegurando su funcionamiento uniforme en distintos entornos (desarrollo, pruebas, producción).
   * Los contenedores son independientes del sistema operativo anfitrión, facilitando la migración de aplicaciones.
2. **Eficiencia en el Uso de Recursos**
   * Los contenedores comparten el kernel del sistema operativo, por lo que no necesitan un sistema operativo completo, como ocurre en las máquinas virtuales (VM).
   * Ocupan menos espacio y optimizan el uso de CPU, memoria y almacenamiento.
3. **Escalabilidad**
   * Docker permite escalar aplicaciones horizontalmente, creando y destruyendo contenedores rápidamente según la demanda.
   * Ideal para aplicaciones de microservicios que requieren escalabilidad dinámica.
4. **Velocidad de Despliegue y Desarrollo**
   * Los contenedores permiten a los desarrolladores construir, probar y desplegar aplicaciones rápidamente, acelerando el ciclo de desarrollo.
   * Reproducir entornos de producción localmente facilita la identificación de problemas.
5. **Facilidad de Automatización y DevOps**
   * Docker se integra fácilmente con herramientas de CI/CD, facilitando la automatización de flujos de trabajo y mejorando la colaboración entre desarrolladores y equipos de operaciones.
6. **Compatibilidad Consistente**
   * Docker elimina el problema de “funciona en mi máquina” encapsulando dependencias y configuraciones necesarias, garantizando consistencia entre entornos.
7. **Aislamiento y Seguridad Mejorada**
   * Los contenedores proporcionan aislamiento de procesos y recursos, reduciendo riesgos de seguridad.
   * Docker permite configurar seguridad específica para cada contenedor.
8. **Ahorro de Costos**
   * Al ser más eficientes que las VM, los contenedores permiten mayor densidad de aplicaciones en el mismo hardware, reduciendo los costos de infraestructura. WEB

**Aplicaciones Modernas: Enfoque Multiservicio y Equipos Especializados**

Las aplicaciones de hoy en día suelen diseñarse como **arquitecturas de múltiples servicios** o microservicios, donde cada componente cumple una función específica dentro del sistema. Cada microservicio puede ser desarrollado, desplegado y mantenido de forma independiente, lo cual permite una gran flexibilidad y escalabilidad.



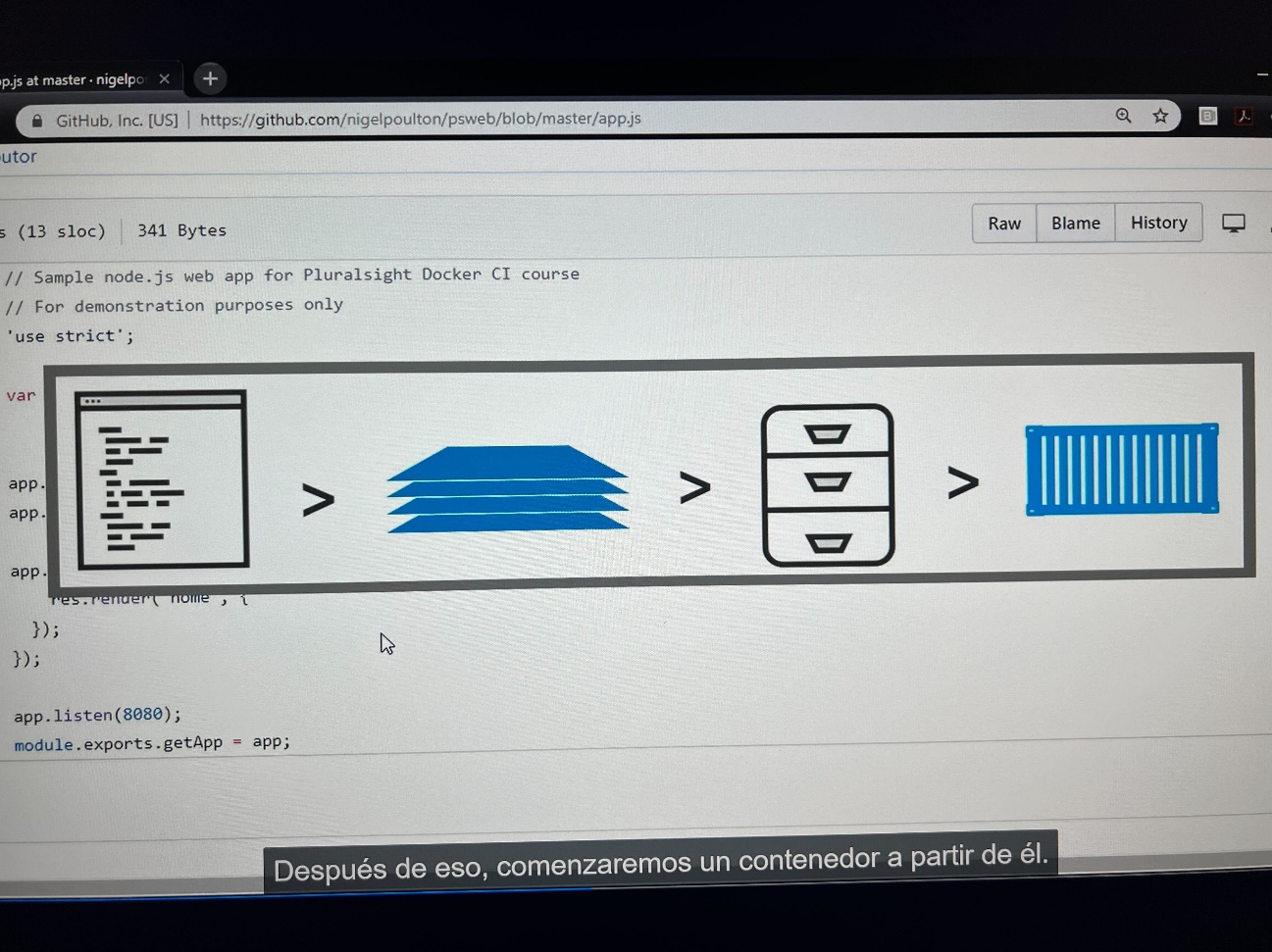
En estos entornos, es común que cada servicio esté a cargo de un equipo diferente, especializado en su funcionalidad y en las tecnologías específicas que requiere. Por ejemplo, una aplicación web moderna puede dividirse en servicios como autenticación, procesamiento de pagos, análisis de datos y manejo de notificaciones, entre otros. Cada equipo puede gestionar y optimizar su propio servicio sin interferir en los demás, lo que acelera el desarrollo y reduce los riesgos de conflictos entre dependencias.

**Docker y Kubernetes en el Entorno Empresarial**

Actualmente, la mayoría de las empresas han adoptado Docker y Kubernetes como estándares para la gestión de aplicaciones en contenedores. Estas herramientas permiten a las organizaciones desplegar, escalar y gestionar sus servicios con mayor eficiencia y control. Solo unas pocas empresas se resisten a esta transformación, pero la tendencia es clara: los contenedores son el camino hacia el futuro. Tarde o temprano, estas empresas también necesitarán adoptar esta tecnología para mantenerse competitivas.

**Consejo**: Prepárate para esta evolución. Familiarizarse con Docker y Kubernetes es una inversión que facilitará trabajar en proyectos modernos y en empresas que apuestan por innovación y eficiencia en sus procesos de despliegue y gestión.

PRACTICA 1- Ejemplo rápido de despliegue de una aplicación (practica hecha por mi)



¿Qué objetivo tenemos?

Estos contenedores, que funcionan como "cajas negras", deben configurarse de acuerdo a los requisitos específicos de cada aplicación. Esto permite que, sin importar dónde se ejecuten (ya sea en Kubernetes, Docker o en una plataforma en la nube), se conviertan en unidades autónomas y completamente integradas.

Cada contenedor puede obtener la información que necesita del entorno donde se despliega y, a su vez, proporciona retroalimentación a la plataforma para asegurar un funcionamiento óptimo.

Este enfoque aporta una gran **consistencia en la operación de las aplicaciones**, garantizando que se ejecuten de manera estable y uniforme en cualquier entorno de despliegue.

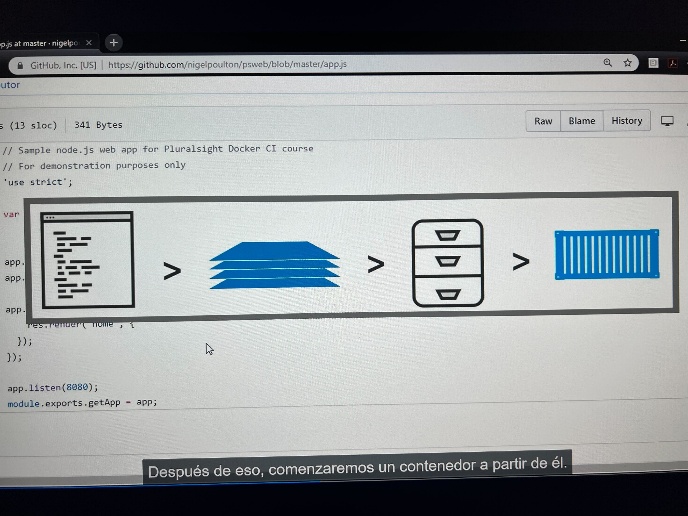
**Arquitectura de Docker**

Docker sigue una estructura cliente-servidor tradicional. En entornos de desarrollo local, tanto el cliente como el servidor Docker suelen estar en la misma máquina, lo que facilita el trabajo y las pruebas. Sin embargo, en entornos de producción, el cliente y el servidor pueden estar separados, y el cliente envía las solicitudes al servidor Docker, que luego ejecuta las operaciones necesarias en los contenedores de la infraestructura.

**Arquitectura de Alto Nivel**

A nivel general, el flujo de trabajo de Docker sigue estos pasos:

1. **Código en cualquier lenguaje**: Partimos del código fuente de la aplicación, sin importar el lenguaje de programación.
2. **Docker Build**: Creamos una imagen utilizando el comando docker build, que genera una imagen siguiendo los estándares de la Open Container Initiative (OCI), organismo que regula las normas de contenedores.
3. **Docker Push**: Subimos la imagen a un repositorio (registro), donde queda almacenada para ser utilizada posteriormente.
4. **Docker Run**: Ejecutamos un contenedor a partir de esta imagen en el servidor, creando un contenedor compatible con el estándar OCI.



**Componentes Clave de Docker**

**Dentro de la arquitectura de alto nivel de Docker, los componentes clave son aquellos que permiten la construcción, administración y ejecución de contenedores de manera eficiente. A continuación se describen estos componentes principales:**

**1. Docker Daemon (dockerd)**

* **Es el proceso principal que se ejecuta en el sistema operativo del host y permite la creación, gestión y monitoreo de contenedores.**
* **Escucha las solicitudes de la CLI de Docker (o de otros clientes) y administra los contenedores, imágenes, volúmenes y redes.**

**2. Docker CLI (Command Line Interface)**

* **La interfaz de línea de comandos (CLI) es la herramienta que permite al usuario comunicarse con el daemon Docker.**
* **Mediante comandos como docker run, docker build, docker pull, etc., los usuarios pueden gestionar contenedores e imágenes.**

**3. Imágenes Docker**

* **Las imágenes son plantillas de solo lectura que contienen el código, las dependencias y configuraciones necesarias para ejecutar una aplicación.**
* **Se construyen a partir de un Dockerfile y consisten en una serie de capas que optimizan el almacenamiento y la gestión de recursos.**

**4. Contenedores Docker**

* **Un contenedor es una instancia en ejecución de una imagen. Son entornos aislados que ejecutan aplicaciones y sus dependencias.**
* **Los contenedores comparten el kernel del sistema operativo del host, lo que los hace más ligeros que las máquinas virtuales y les permite un arranque rápido.**

**5. Dockerfile**

* **Es el archivo de configuración que contiene las instrucciones para construir una imagen Docker.**
* **Los comandos en un Dockerfile definen las capas de la imagen, incluyendo la instalación de paquetes, copiado de archivos y configuración de variables de entorno.**

**6. Registries (Registro de Imágenes)**

* **Los registros son servidores donde se almacenan y gestionan las imágenes de Docker.**
* **Docker Hub es el registro público por defecto, pero también se pueden configurar registros privados para mantener imágenes personalizadas o sensibles.**

**7. Redes Docker**

* **Docker permite crear redes para facilitar la comunicación entre contenedores.**
* **Los principales tipos de redes incluyen bridge (red por defecto en Docker), host, overlay y none. Cada red tiene características que se ajustan a diferentes casos de uso, como redes privadas entre contenedores o redes para conectar múltiples hosts.**

**8. Volúmenes Docker**

* **Los volúmenes se usan para almacenar datos de forma persistente y compartir datos entre contenedores o entre un contenedor y el host.**
* **A diferencia del almacenamiento temporal de los contenedores, los volúmenes permiten que los datos persistan incluso si el contenedor es eliminado.**

**9. Docker Compose**

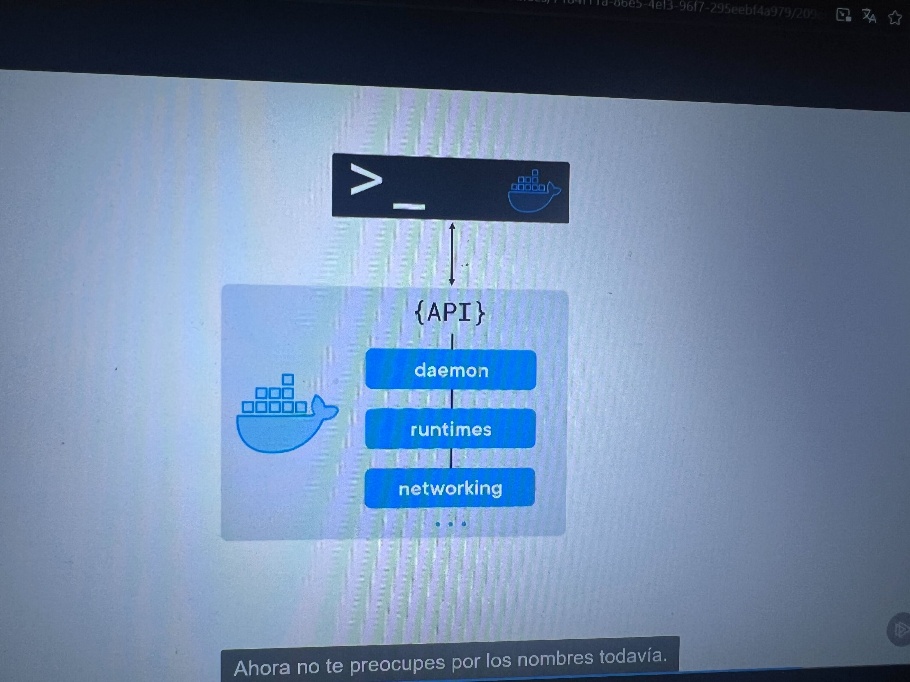
* **Es una herramienta para definir y ejecutar aplicaciones multi-contenedor mediante un archivo docker-compose.yml.**
* **Permite crear, configurar y orquestar múltiples servicios en una sola aplicación, lo que facilita la administración y escalabilidad.**

**UN poco de historia**

Un contenedor es un entorno aislado que ejecuta un proceso dentro de una sección dedicada del sistema operativo. Los contenedores usan dos mecanismos del kernel de Linux para proporcionar aislamiento:

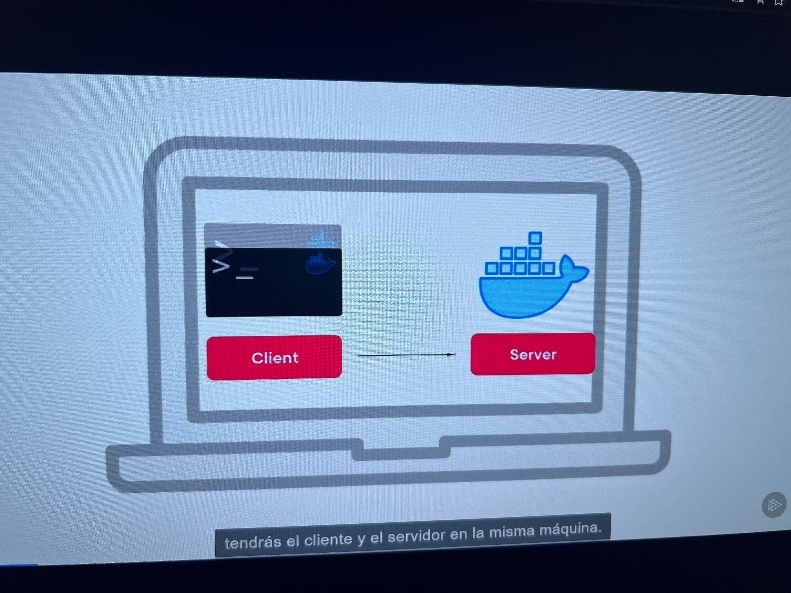
* + **Namespaces**: Dividen los recursos del sistema para garantizar que el contenedor opere de forma independiente de otros procesos y del sistema anfitrión.
  + **Control Groups (cgroups)**: Establecen límites de recursos para cada contenedor, como CPU, memoria y red.

Aunque estos conceptos existen desde hace tiempo en el kernel de Linux, Docker los simplificó y los hizo accesibles a los desarrolladores a través de una interfaz de línea de comandos (CLI). Esta CLI permite ejecutar comandos de Docker de forma sencilla, aunque, internamente, hay una serie de procesos complejos que gestionan la ejecución y el aislamiento de cada contenedor.



**Internos del Kernel y Comparación con Virtualización**

En el caso de un hipervisor, se divide una máquina física en múltiples máquinas virtuales (VM), cada una con su propio sistema operativo y un conjunto específico de recursos (CPU, memoria, etc.). En cambio, los contenedores Docker comparten el mismo sistema operativo del anfitrión, utilizando una interfaz de red común (como eth0), un sistema de archivos compartido y un único árbol de procesos. Aunque varios contenedores pueden operar en paralelo, todos comparten el kernel del sistema operativo del anfitrión, lo cual los hace mucho más ligeros y eficientes en comparación con las máquinas virtuales.



A computer screen with a computer screen and a computer screen

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

PRACTICA 2 -> Jugando con Docker (practica en un word)

**¿Qué es una Imagen en Docker?**

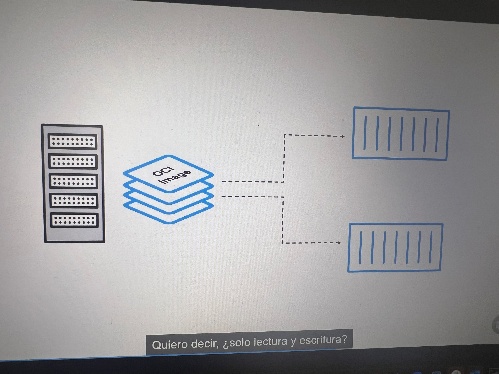
Una **imagen** en Docker es una plantilla de solo lectura que contiene todo el código de la aplicación, las bibliotecas y los archivos de soporte necesarios para ejecutarla. Esta capacidad de encapsular todo lo necesario en una única imagen ha revolucionado la forma de desplegar aplicaciones, eliminando problemas de compatibilidad entre entornos con diferentes bibliotecas o versiones.

Básicamente, una imagen es como un “contenedor detenido”, lista para ejecutarse cuando se necesite.

**Creación de Imágenes y su Estructura en Capas**

Las imágenes se construyen en **capas apiladas** del sistema de archivos. Este modelo de capas es muy eficiente, ya que permite:

* Iniciar múltiples contenedores desde la misma imagen, lo que ahorra espacio.
* Compartir capas entre diferentes imágenes cuando contienen elementos comunes, optimizando aún más el almacenamiento.



A diagram of a diagram

Description automatically generated with medium confidence

Normalmente, las imágenes se almacenan en un **registro** (como Docker Hub) y pueden descargarse (hacer “pull”) cuando se necesitan. En un sistema donde varios contenedores usan la misma imagen, las capas de esta imagen son de solo lectura, asegurando que los datos permanezcan constantes y no se vean alterados.

**Escritura en Contenedores**

Aunque la imagen es de solo lectura, cada contenedor tiene su propia **capa de escritura temporal** donde se almacenan datos específicos de su ejecución. Esta capa se apila sobre las capas de la imagen. Sin embargo, cualquier dato almacenado aquí es volátil y se pierde al detener el contenedor.

Para datos persistentes que deben sobrevivir entre ejecuciones, se usan **volúmenes** de Docker. Estos permiten que los contenedores escriban y accedan a información de manera permanente.

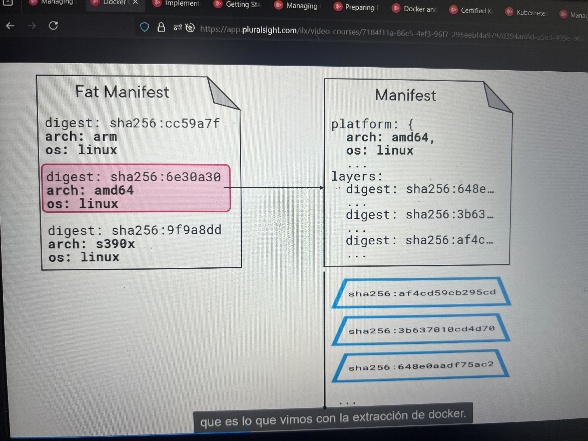
**Manifiesto de una Imagen**

Cada imagen tiene un **manifiesto**, un archivo de configuración que describe la estructura de la imagen:

* ID de la imagen
* Fecha de creación
* Lista de capas y su orden

CAPAS + MANIFIESTO = IMAGEN

Este manifiesto permite apilar las capas de la imagen en el orden correcto y adapta la imagen a diferentes arquitecturas si es necesario. A veces, un manifiesto mayor referencia otros manifiestos específicos de cada arquitectura, para que Docker descargue solo los elementos necesarios.



**Controlador de Almacenamiento de Imágenes**

Detrás de las capas y el manifiesto, Docker usa un **controlador de almacenamiento** que gestiona cómo se organizan las capas en el sistema de archivos. Para observar el controlador de almacenamiento en uso, puedes ejecutar docker info y buscar el campo **Storage Driver**.

**Ruta de Almacenamiento**:

* En Linux: /var/lib/docker
* En Windows: C:\ProgramData\Docker

PRACTICA SEGURIDAD: Comprobar que la img que nos hemos descargado es esa

1. Pull de una imagen. Aparece el digest
2. Docker images –digests
3. Comprobar que ambos son iguales

Resumiendo: tenemos las capas, vinculadas y apiladas por un manifiesto y algún sistema crptográfico que nos da mutabilidad.

**Buenas Prácticas para el Uso de Imágenes en Docker**

1. **Usa imágenes oficiales siempre que sea posible**: Las imágenes oficiales en Docker Hub están verificadas y seguras.
2. **Mantén las imágenes lo más ligeras posible**: Esto reduce el tiempo de descarga y el uso de recursos.
3. **Construye imágenes personalizadas solo cuando sea necesario**: Si una imagen oficial no cumple los requisitos, puedes construir la tuya a partir de una imagen base pequeña y confiable.
4. **Usa versiones exactas en lugar de latest**: Especificar versiones exactas evita actualizaciones inesperadas y asegura la estabilidad en los entornos de producción.

A menudo, necesitamos configurar una imagen personalizada para ejecutar una aplicación con las configuraciones específicas requeridas. El objetivo es encapsular la aplicación dentro de una imagen Docker que pueda desplegarse en cualquier entorno. Para lograr esto, creamos un **Dockerfile**, que contiene todas las instrucciones necesarias para construir la imagen.

**Proceso**:

APP + Dockerfile -> imagen (Docker build) -> contenedor ejecutando app (docker run)

**Descargar una App de Prueba**: Crear un repo con una app sencilla pedir a los participantes que descarguen una aplicación sencilla.

 **Revisión del Dockerfile**: Mostrar un ejemplo de Dockerfile del proyecto (una foto o archivo adjunto es útil aquí). Es importante resaltar que un Dockerfile bien escrito es fácil de entender y contribuye a la mantenibilidad de la aplicación.

* **Explicación del Dockerfile**: Revisa las instrucciones del Dockerfile, aclarando que cada instrucción crea una nueva capa en la imagen y que el Dockerfile en sí es solo una configuración de cómo se construirá la imagen.

**2. Construcción Multi-etapa (Multi-Stage Building)**

El **multi-stage building** es una técnica avanzada en Docker que permite crear imágenes optimizadas y más ligeras, eliminando las dependencias y archivos que solo se necesitan durante la construcción. En una construcción multi-etapa, se pueden usar varias etapas en el Dockerfile para realizar el build en una etapa y luego copiar solo los archivos necesarios a la imagen final.

# Primera etapa: build

FROM golang:1.16 AS builder

WORKDIR /app

COPY . .

RUN go build -o main .

# Segunda etapa: solo ejecutable

FROM alpine:latest

COPY --from=builder /app/main /app/

CMD ["/app/main"]

Esta práctica de construcción reduce el tamaño de la imagen final y elimina archivos innecesarios.

**Buenas Prácticas para la Creación de Dockerfiles**

1. **Imagen Base (Base Image)**
   * Toda imagen debe comenzar con un FROM, preferiblemente desde una imagen oficial o un registry de confianza. Utiliza imágenes ligeras (como alpine) para reducir vulnerabilidades y minimizar el sistema de archivos.
2. **Usar Multi-Stage Building**
   * Aplicar multi-stage building para eliminar contenido innecesario y reducir la superficie de ataque.
3. **Configurar Usuarios Sin Privilegios**
   * Añadir un usuario sin privilegios en lugar de ejecutar el contenedor como root.
4. **Utilizar Volúmenes Solo para Persistencia Necesaria**
   * Usa volúmenes para la persistencia, pero evita montar directorios sensibles como /etc o /bin.
5. **Configurar el Sistema de Archivos en Modo de Solo Lectura**
   * Configura el sistema de archivos como solo lectura, excepto donde se necesite persistencia.
6. **Precaución con el Manejo de Secretos**
   * Evita pasar secretos directamente en el Dockerfile. Usa herramientas de gestión de secretos seguras.

Practicas guiadas:

PRACTICA usuarios con privilegios: <https://app.pluralsight.com/ilx/video-courses/9a6f3b2c-69d2-428c-908b-88d1a2431fe8/1da49361-2979-4b1d-911a-41bedac6b0d1/a7aea1f0-cdba-45a5-bf69-950c718fe3c4>

PRACTICA – configuración peligrosa : <https://app.pluralsight.com/ilx/video-courses/9a6f3b2c-69d2-428c-908b-88d1a2431fe8/1da49361-2979-4b1d-911a-41bedac6b0d1/1a0faaf8-60a7-4a8e-a8a2-2b3145581ee7>

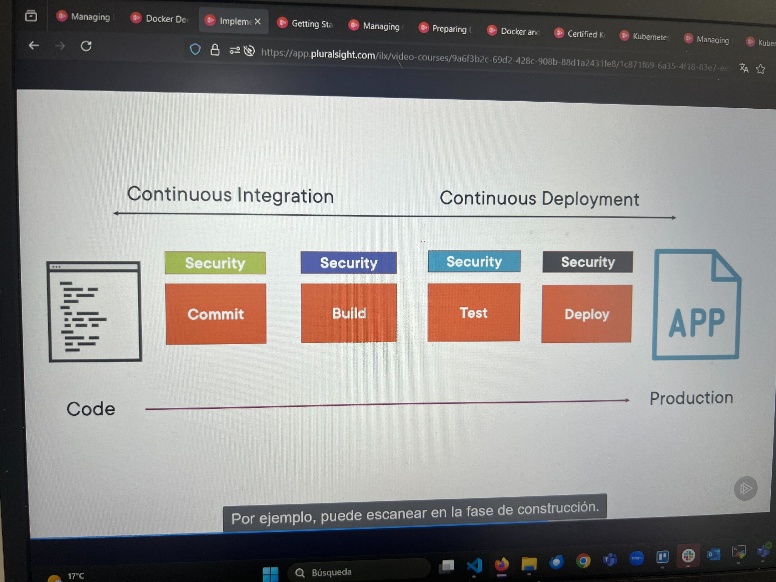
**Seguridad en Imágenes Docker**

Docker utiliza capas en la construcción de imágenes, y cuando el contenedor se ejecuta, añade una capa de escritura sobre las capas de solo lectura de la imagen. Es importante tener en cuenta los vectores de ataque potenciales en imágenes Docker:

1. **Modificación del Código Fuente**: Un atacante podría alterar el código de la aplicación.
2. **Dockerfile Manipulado**: Cambiar instrucciones en el Dockerfile puede exponer secretos o introducir vulnerabilidades.
3. **Vulnerabilidades en la Imagen Base**: Las imágenes base deben mantenerse actualizadas y verificadas para evitar que contengan vulnerabilidades.
4. **Ataques a la Máquina de Construcción**: El entorno de construcción debe ser seguro, eliminando herramientas innecesarias y aislándolo del entorno de producción.
5. **Registro Comprometido**: Las imágenes deben almacenarse en un registro seguro y verificado.

**Práctica Recomendada**:

* Configurar y asegurar la máquina que genera las imágenes, eliminando herramientas y manteniéndola aislada del entorno de producción.
* Ejecutar chequeos de seguridad en el pipeline de CI/CD mediante escaneos automáticos



**Herramientas de Escaneo**:

* **Dockle** y **Trivy** son herramientas populares que realizan análisis de seguridad en imágenes Docker y ayudan a identificar vulnerabilidades antes del despliegue.

**Práctica**: Uso de Dockle y Trivy para realizar análisis de seguridad en imágenes Docker, y verificar el cumplimiento de las mejores prácticas. Creando imágenes seguras <https://app.pluralsight.com/ilx/video-courses/9a6f3b2c-69d2-428c-908b-88d1a2431fe8/1c871f69-6a35-4f18-83e7-ee51c80b5ae4/50d82482-7595-45a8-bbe8-958e40d15f26>

**Consideraciones Adicionales**

Además de asegurar la imagen Docker, es importante securizar el entorno Docker completo:

1. **Protección del Docker Daemon**: Configura el Daemon de Docker para que solo acepte conexiones seguras.
2. **Seguridad en el Docker Host**: Asegura el sistema operativo donde corre Docker siguiendo las recomendaciones del CIS y aplicando escaneos de seguridad periódicos.

Estas prácticas ayudan a mantener un entorno Docker seguro y minimizan riesgos en entornos de producción.

**Contenedores en Docker**

En Docker, el **contenedor** es la unidad de trabajo más pequeña que podemos implementar. Los contenedores son instancias activas de una imagen y representan entornos de ejecución aislados donde se ejecuta una aplicación específica.

Podemos realizar varias acciones con los contenedores: **iniciarlos, detenerlos, reiniciarlos o eliminarlos**. Hasta que se eliminen, los contenedores mantienen su estado, permitiendo conservar la configuración y los datos temporales generados durante su ejecución.

Es importante destacar que, en el contexto de Docker, cuando hablamos de contenedores, nos referimos a aplicaciones. Cada contenedor está diseñado para ejecutar un único proceso o servicio, manteniendo una única responsabilidad. Este enfoque simplifica la administración y asegura una arquitectura modular.

**Resumen de los Aspectos Clave de los Contenedores**

* **Entornos de Ejecución**: Cada contenedor tiene su propio árbol de procesos, sistema de archivos y entorno de red, lo cual le proporciona aislamiento y evita interferencias con otros contenedores.
* **Acciones Disponibles**: Los contenedores pueden iniciarse, detenerse, borrarse y reiniciarse según sea necesario.
* **Naturaleza Efímera**: Los contenedores suelen ser efímeros y están diseñados para ser actualizados, escalados y reemplazados sin afectar la aplicación en general.
* **Inmutabilidad**: No se recomienda modificar el contenido dentro de un contenedor en ejecución. En su lugar, es mejor construir u

PRACTICA: Arranque y exposición de un contenedor

VER PRACTICA EN MD

**Networking en Docker**

Docker permite conectar contenedores a redes virtuales para que puedan comunicarse entre sí. Los principales tipos de redes en Docker son:

* **Bridge**: Red por defecto en Docker. Los contenedores en una misma red bridge pueden comunicarse entre ellos, pero están aislados del resto.
* **Host**: El contenedor comparte la red del host, eliminando el aislamiento.
* **Overlay**: Para conectar contenedores en diferentes hosts (usado en Swarm o Kubernetes).
* **None**: Sin red; el contenedor está completamente aislado.

**Comando clave**: docker network create para crear redes personalizadas.

**Persistencia en Docker**

La persistencia permite conservar datos generados por los contenedores. Docker usa **volúmenes** y **bind mounts** para lograr esto:

* **Volúmenes**: Son gestionados por Docker y recomendados para datos que deben persistir independientemente de la ubicación del contenedor.
* **Bind mounts**: Montan directorios específicos del host en el contenedor; útiles para desarrollo pero menos portables.

**Comando clave**: docker volume create para crear un volumen.

**Docker Compose: Networking y Persistencia en Multi-Contenedor**

Docker Compose es una herramienta que facilita la definición y gestión de aplicaciones multi-contenedor en un solo archivo (docker-compose.yml). Con Docker Compose puedes:

* **Definir redes y volúmenes**: Crear redes y volúmenes compartidos entre servicios.
* **Automatizar despliegues**: Iniciar, detener y escalar contenedores de manera conjunta.

**Exposición de Puertos en Docker**

Para que los servicios de los contenedores sean accesibles desde fuera de Docker, se deben **exponer puertos** en el contenedor y mapearlos a puertos del host. Esto se hace con la opción -p en docker run o en Docker Compose usando la clave ports.

* **Sintaxis**: -p <puerto\_host>:<puerto\_contenedor>
  + Por ejemplo, -p 8080:80 hace que el puerto 80 del contenedor sea accesible en el puerto 8080 del host.

Practica en md en el ordenador Docker\_Compose\_Practice\_phpMyAdmin\_MySQL