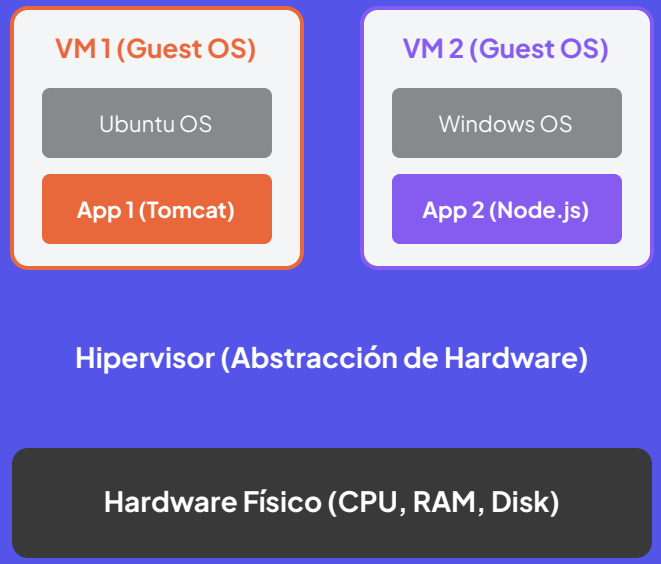
Universidad Icesi

Virtualización y Contenedores con Docker

Estandarización, Despliegue e Infraestructura Inmutable

Computación en Internet 3 — Universidad Icesi

Universidad ICESI



¿Qué es la Virtualización?

La **Virtualización** es una tecnología que permite crear representaciones basadas en software (virtuales) de recursos de hardware físico.

- **El Hipervisor:** Capa de software (ej. VMware, VirtualBox, KVM) que abstrae la CPU, RAM y almacenamiento físico del servidor.
- **Máquinas Virtuales (VMs):** Ambientes aislados que ejecutan un Sistema Operativo invitado completo (Guest OS).
- **Propósito Histórico:** Consolidar múltiples servidores físicos en una sola máquina física para aprovechar el hardware inactivo.

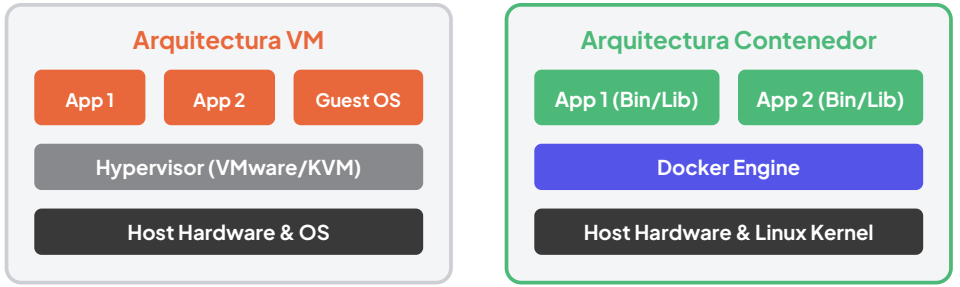
Virtualización vs. Contenedores

Máquinas Virtuales (VMs)

- **Simulación Hardware:** Cada VM ejecuta un Sistema Operativo invitado completo (Guest OS).
- **Consumo de Recursos:** Requiere giga-bytes de memoria RAM y CPU reservada por máquina.
- **Lentitud:** Minutos para arrancar la máquina virtual e iniciar los servicios.

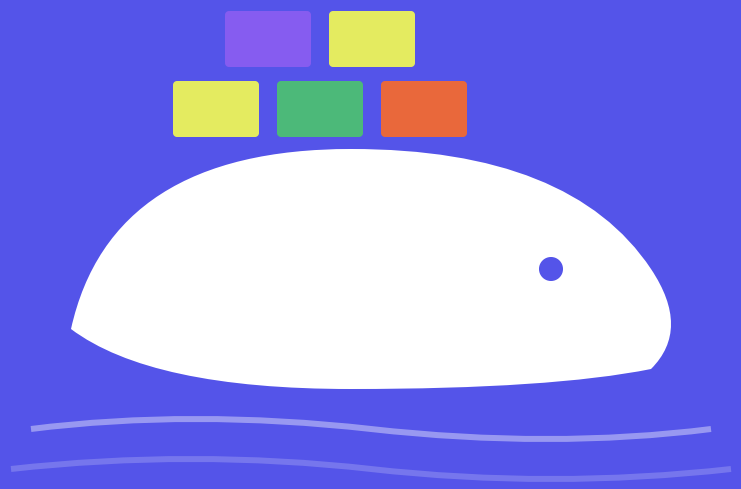
Contenedores Docker

- **Kernel Compartido:** Comparten el mismo kernel de Linux del sistema anfitrión (Host OS).
- **Aislamiento Liviano:** Aísla procesos a nivel del SO sin simular hardware.
- **Ultrarrápido:** Inicia aplicaciones aisladas en cuestión de milisegundos.



Universidad ICESI

Standardized Containers



Docker: ¿Qué es?

Docker es una plataforma de software abierta que encapsula el proceso de empaquetar, distribuir y ejecutar aplicaciones dentro de contenedores estandarizados.

- **Artefacto Distribuible:** Empaca el código de la app, sus binarios, librerías y configuraciones en una sola unidad sellada.
- **Despliegue a Gran Escala:** Garantiza que el artefacto funcione idénticamente en laptop, servidor físico o la nube.
- **Metáfora Marítima:** Al igual que el contenedor de carga estandarizó el transporte marítimo mundial, Docker estandariza el transporte de software.

La Promesa de Docker: ¿Qué busca ofrecernos?

01. PACKAGING SOFTWARE

Empaquetar software aprovechando las habilidades que los desarrolladores ya poseen, sin necesidad de reescribir la infraestructura.

La Promesa de Docker: ¿Qué busca ofrecernos?

01. PACKAGING SOFTWARE

Empaquetar software aprovechando las habilidades que los desarrolladores ya poseen, sin necesidad de reescribir la infraestructura.

02. SINGLE STANDARDIZED IMAGE

Construir la aplicación y los sistemas de archivos del SO requeridos juntos en un único formato de imagen estandarizado.

La Promesa de Docker: ¿Qué busca ofrecernos?

01. PACKAGING SOFTWARE

Empaquetar software aprovechando las habilidades que los desarrolladores ya poseen, sin necesidad de reescribir la infraestructura.

02. SINGLE STANDARDIZED IMAGE

Construir la aplicación y los sistemas de archivos del SO requeridos juntos en un único formato de imagen estandarizado.

03. TEST & DELIVER EXACTLY

Usar los artefactos empaquetados para probar y entregar el **mismo artefacto exacto** en todos los entornos.

Universidad ICESI

La Promesa de Docker: ¿Qué busca ofrecernos?

01. PACKAGING SOFTWARE

Empaquetar software aprovechando las habilidades que los desarrolladores ya poseen, sin necesidad de reescribir la infraestructura.

02. SINGLE STANDARDIZED IMAGE

Construir la aplicación y los sistemas de archivos del SO requeridos juntos en un único formato de imagen estandarizado.

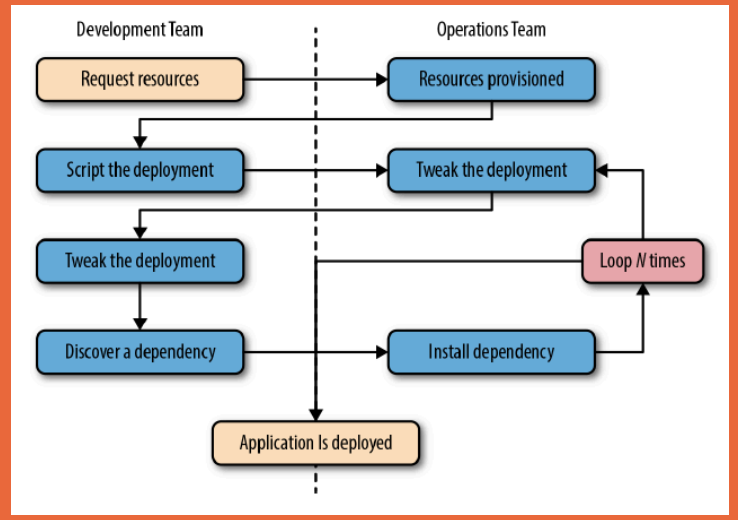
03. TEST & DELIVER EXACTLY

Usar los artefactos empaquetados para probar y entregar el **mismo artefacto exacto** en todos los entornos.

04. HARDWARE ABSTRACTION

Abstraer las aplicaciones del hardware subyacente sin sacrificar recursos del sistema ni incurrir en sobrecostos.

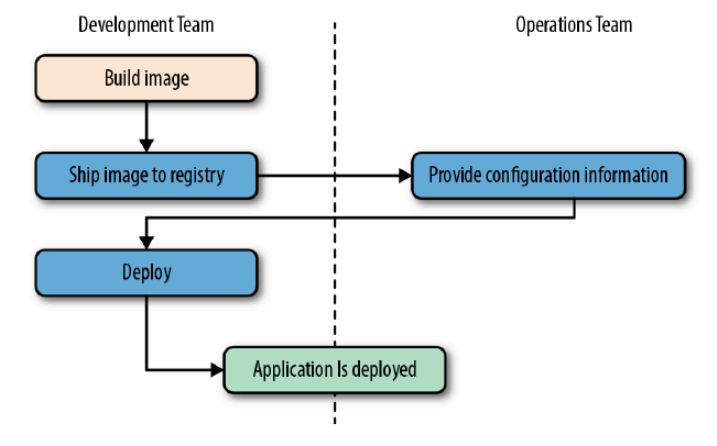
El Flujo Tradicional (Sin Docker)



Antes de los contenedores, el despliegue de software enfrentaba constante fricción y demoras:

- **Fricción Dev vs Ops:** Fuerte dependencia en comunicación manual entre equipos de desarrollo y operaciones.
- **Extrema Lentitud:** Configurar un servidor y desplegar una app tomaba casi una semana.
- **El Síndrome "En mi máquina funciona":** Descubrimiento tardío de dependencias o versiones distintas de librerías en producción.

El Flujo de Desarrollo con Docker



Docker introduce una separación limpia y desacoplada de responsabilidades:

- **Desarrollo construye la Imagen:** Empaca la app y la sube al Registro central de imágenes.
- **Operaciones provee Infraestructura:** Descarga la imagen y asigna recursos para su ejecución.
- **Punto de Entrega Pacífico:** El *Registry* se convierte en el contrato inmutable de entrega.

Aclarando Mitos: Qué NO es Docker

NO ES VM ENTERPRISE

No es virtualización tradicional. Los contenedores no emulan hardware ni corren kernels independientes por cliente.

Aclarando Mitos: Qué NO es Docker

NO ES VM ENTERPRISE

No es virtualización tradicional. Los contenedores no emulan hardware ni corren kernels independientes por cliente.

NO ES DEPLOYMENT FRAMEWORK

No automatiza despliegues complejos por sí solo. Empaca dependencias, pero requiere orquestadores a gran escala.

Aclarando Mitos: Qué NO es Docker

NO ES VM ENTERPRISE

No es virtualización tradicional. Los contenedores no emulan hardware ni corren kernels independientes por cliente.

NO ES DEPLOYMENT FRAMEWORK

No automatiza despliegues complejos por sí solo. Empaca dependencias, pero requiere orquestadores a gran escala.

NO ES DEV ENVIRONMENT

No reemplaza totalmente a herramientas tipo Vagrant. No simula redes/servidores completos en local.

Aclarando Mitos: Qué NO es Docker

NO ES VM ENTERPRISE

No es virtualización tradicional. Los contenedores no emulan hardware ni corren kernels independientes por cliente.

NO ES DEPLOYMENT FRAMEWORK

No automatiza despliegues complejos por sí solo. Empaca dependencias, pero requiere orquestadores a gran escala.

NO ES DEV ENVIRONMENT

No reemplaza totalmente a herramientas tipo Vagrant. No simula redes/servidores completos en local.

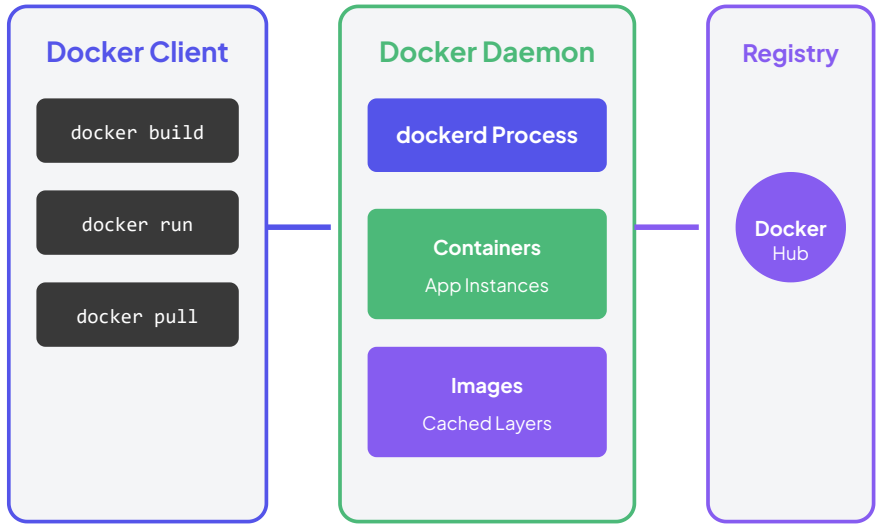
ENFOQUE REAL DOCKER

Es un motor de empaquetado inmutable y aislamiento liviano de procesos para garantizar consistencia universal.

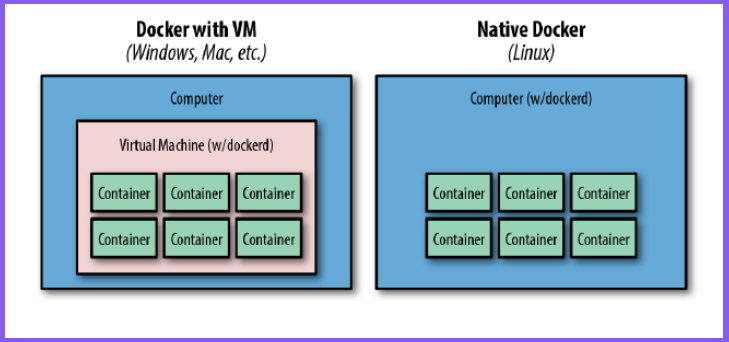
Partes de Docker: Arquitectura Cliente-Servidor

Docker utiliza una arquitectura desacoplada Cliente-Servidor basada en APIs REST:

- **1. Docker Client:** El comando CLI (`docker`) utilizado por el desarrollador para controlar el flujo de trabajo.
- **2. Docker Server (Daemon):** El proceso en segundo plano `dockerd` que hace el trabajo pesado de construir, ejecutar y administrar contenedores.
- **3. Docker Registry:** Almacena y distribuye imágenes y metadatos (ej. Docker Hub).



Universidad ICESI



Soporte Multiplataforma

Aunque Docker es tecnología nativa de Linux, sus herramientas oficiales permiten trabajar en cualquier sistema:

- **Linux Nativo:** El daemon `dockerd` corre directamente sobre el kernel de Linux.
- **macOS & Windows:** Instalan una máquina virtual liviana (Hyper-V / WSL2 / xhyve) en segundo plano.
- **Transparente para el Dev:** El cliente CLI envía comandos exactamente igual sin importar el SO del host.

Principios de Diseño con Contenedores

CONTENEDORES LIGEROS

Un contenedor nuevo puede requerir solo ~12 KB de espacio adicional, ya que usa capas compartidas sin duplicar el SO.

Principios de Diseño con Contenedores

CONTENEDORES LIGEROS

Un contenedor nuevo puede requerir solo ~12 KB de espacio adicional, ya que usa capas compartidas sin duplicar el SO.

IMMUTABLE INFRASTRUCTURE

Los componentes se reemplazan enteros en lugar de modificarse en caliente, simplificando la gestión.

Principios de Diseño con Contenedores

CONTENEDORES LIGEROS

Un contenedor nuevo puede requerir solo ~12 KB de espacio adicional, ya que usa capas compartidas sin duplicar el SO.

IMMUTABLE INFRASTRUCTURE

Los componentes se reemplazan enteros en lugar de modificarse en caliente, simplificando la gestión.

STATELESS APPLICATIONS

Las aplicaciones procesan peticiones sin guardar datos locales entre sesiones en el disco interno.

Principios de Diseño con Contenedores

CONTENEDORES LIGEROS

Un contenedor nuevo puede requerir solo ~12 KB de espacio adicional, ya que usa capas compartidas sin duplicar el SO.

IMMUTABLE INFRASTRUCTURE

Los componentes se reemplazan enteros en lugar de modificarse en caliente, simplificando la gestión.

STATELESS APPLICATIONS

Las aplicaciones procesan peticiones sin guardar datos locales entre sesiones en el disco interno.

EXTERNALIZING STATE

El estado y almacenamiento persistente se debe delegar a bases de datos externas o volúmenes montados.

Universidad ICESI

Hands On Docker

De la Teoría a la Práctica

Construcción de Artefactos, Inspección de Capas y Control del Ciclo de Vida

Bloques Fundamentales de Docker

01. IMÁGENES

Plantillas de solo lectura compuestas por capas de sistema de archivos.

02. CONTENEDORES

Instancia aislada y en ejecución (o detenida) instanciada desde una imagen.

03. VOLÚMENES

Mecanismo para persistir datos fuera del ciclo efímero del contenedor.

04. PUERTOS & RED

Canal de comunicación que mapea sockets del host con el contenedor.

Anatomía de una Imagen: Capas y Dockerfile

Cada instrucción en un `Dockerfile` genera una capa nueva en el sistema de archivos (UnionFS):

- `FROM` : Capa base del SO o runtime (ej. Node, Python, Alpine).
- `COPY package*.json` : Separa la copia de manifiestos de dependencias para maximizar el uso de la caché.
- `RUN npm install` : Ejecuta comandos y guarda el resultado en una capa inmutable de librerías.
- `COPY . .` : Añade el código fuente de la aplicación en la capa superior.

Dockerfile Sencillo (Node.js API)

```
# Capa 1: Imagen Base
FROM node:18-alpine
WORKDIR /app

# Capa 2: Manifiesto de dependencias
COPY package*.json ./
RUN npm install

# Capa 3: Código fuente
COPY . .

# Metadatos de ejecución
EXPOSE 3000
CMD ["node", "server.js"]
```

Universidad ICESI

Imagen Tradicional Monolítica

App + JDK + Maven + Código Fuente (~750 MB)

Imagen Multi-Stage Optimizada

Solo JRE + Artefacto JAR Compilado (~110 MB)

Optimización de Imágenes & Multi-Stage Builds

El Problema del Tamaño de Imagen

Al construir aplicaciones en lenguajes compilados (Java, C++, Go), se necesitan SDKs y herramientas de build (Maven, Gradle, GCC) que ocupan cientos de MB y vulneran la seguridad en producción.

La Solución: Multi-Stage Builds

- Permite declarar múltiples bloques `FROM` en un único archivo `Dockerfile`.
- Se compila el artefacto en una primera etapa pesada y se copia **únicamente el binario ejecutable** a una segunda etapa ultra-liviana.

Ejemplo de Multi-stage Build (Java Spring Boot)

```
# Etapa 1: Build de la aplicación con Maven
FROM maven:3.9.8-eclipse-temurin-17-alpine AS build
WORKDIR /app

# Copiar dependencias y compilar
COPY pom.xml .
RUN mvn dependency:go-offline -B
COPY src ./src
RUN mvn package -DskipTests

# Etapa 2: Imagen liviana de ejecución
FROM eclipse-temurin:17-jre-alpine
WORKDIR /app

# Copiar solo el JAR generado desde build
COPY --from=build /app/target/*.jar app.jar

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Análisis Sintáctico

- `AS build`: Asigna el nombre o alias *build* a la primera etapa de compilación.
- `FROM eclipse-temurin...`: Reinicia el contexto del sistema de archivos con una imagen limpia sin herramientas de build.
- `COPY --from=build`: Transfiere el archivo `app.jar` desde la etapa *build* hacia la imagen final.

Anatomía y Flujo de Multi-stage Build

El flujo de ejecución de un Multi-stage build garantiza seguridad e inmutabilidad:

- **Fase 1 – Construcción Efímera:** Se descargan dependencias y se compila el ejecutable. Esta capa intermedia se desecha.
- **Fase 2 – Transferencia de Artefacto:** Únicamente los archivos resultantes necesarios para ejecutar se copian a la imagen final.
- **Resultado:** El contenedor en producción no tiene Maven, ni código fuente, ni compiladores instalados.

Construyendo la Imagen: docker build

El comando `docker build` procesa el `Dockerfile` y empaqueta el artefacto en el daemon local:

Sintaxis del Comando:

```
docker build -t springboot-app:v1 .
```

- `-t springboot-app:v1`: Etiqueta (tag) la imagen con un nombre amigable y versión.
- `.`: Define el *contexto de construcción* (directorio actual enviado al daemon).

Salida Simulada de la Terminal:

```
$ docker build -t springboot-app:v1 .
[+] Building 12.4s (12/12) FINISHED
 => [internal] load build definition from Dockerfile
 => [build 1/5] FROM docker.io/library/maven:3.9.8
 => [build 4/5] COPY src ./src
 => [build 5/5] RUN mvn package -DskipTests
 => [stage-1 3/3] COPY --from=build /app/target/*.jar
 => exporting to image
 => => naming to docker.io/library/springboot-app:v1

Successfully built 8e9f2a1b4c
Successfully tagged springboot-app:v1
```

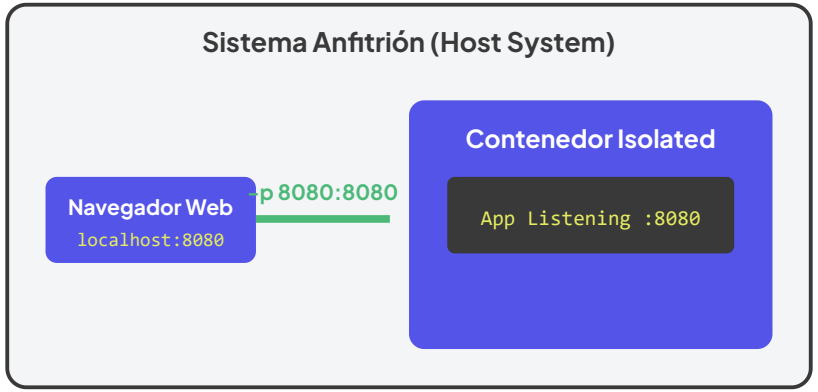
Ejecutando el Contenedor: docker run

El comando `docker run` instancia y ejecuta un contenedor a partir de una imagen previa:

```
docker run -d -p 8080:8080 --name springboot-app springboot-app:v1
```

- `-d` : Detached mode (ejecuta en segundo plano liberar la terminal).
- `-p 8080:8080` : Mapea el puerto del Host al puerto del Contenedor (`Host:Container`).
- `--name` : Asigna un nombre personalizado para identificar el contenedor.

Esquema de Mapeo de Puertos

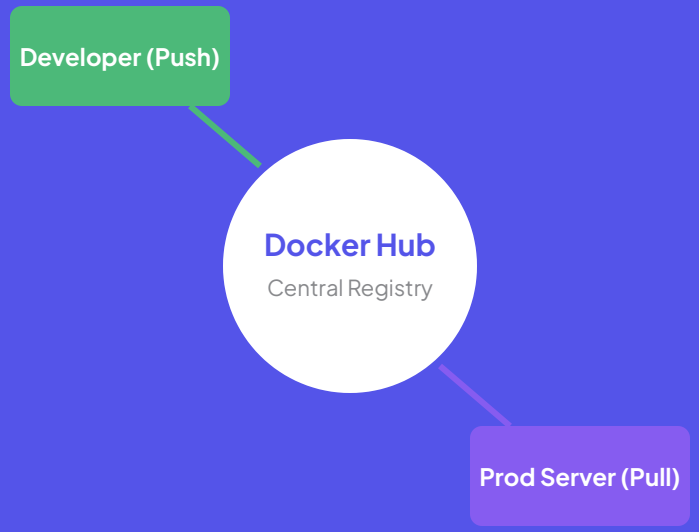


Comandos Útiles del Día a Día: Imágenes y Contenedores

Comando	Descripción y Uso Principal
`docker images`	Lista todas las imágenes almacenadas localmente en la caché del daemon.
`docker pull <imagen>:<tag>`	Descarga una imagen específica desde un repositorio remoto (ej. Docker Hub).
`docker rmi <imagen>:<tag>`	Elimina una imagen descargada del sistema local para liberar espacio.
`docker create --name <nombre> <img>`	Crea un nuevo contenedor a partir de una imagen sin iniciarlo inmediatamente.
`docker start <id/nombre>`	Inicia la ejecución de uno o varios contenedores previamente creados o detenidos.
`docker run <id/nombre>`	Crea e inicia un contenedor en un solo paso (equivalente a `create` + `start`).
`docker rm <nombre-contenedor>`	Elimina un contenedor detenido del sistema local.

Comandos Útiles del Día a Día: Control, Logs y Redes

Comando	Descripción y Uso Principal
`docker stop <id-contenedor>`	Detiene la ejecución de un contenedor enviando la señal `SIGTERM`.
`docker ps -a`	Muestra el listado de todos los contenedores (en ejecución y detenidos).
`docker logs --follow <nombre>`	Muestra la salida estándar de logs del contenedor en tiempo real (modo *tail*).
`docker network ls`	Lista todas las redes virtuales configuradas en el daemon de Docker.
`docker network create <nombre-red>`	Crea una nueva red virtual (tipo *bridge*) para comunicar contenedores.
`docker network rm <nombre-red>`	Elimina una red virtual existente que no esté en uso.
`docker build -t <nombre>:<tag> .`	Construye una imagen de Docker a partir de las instrucciones de un `Dockerfile`.



Registry: Almacenamiento y Distribución

Un **Registry** es un repositorio central para almacenar, guardar versiones y distribuir imágenes de Docker.

- **Docker Hub:** Registro público y oficial por defecto proporcionado por Docker.
- **Desacoplamiento:** Separa las responsabilidades de construcción (Build) y despliegue (Deploy).
- **Punto de Entrega:** Desarrollo sube la imagen al registro y producción la descarga e instancia.

Compartiendo Imágenes: Tag, Push y Pull

El flujo de distribución hacia un Registry remoto sigue 3 pasos principales:

1. Etiquetar (Tag):

```
docker tag springboot-app:v1 kelocoes/springboot-app:v1
```

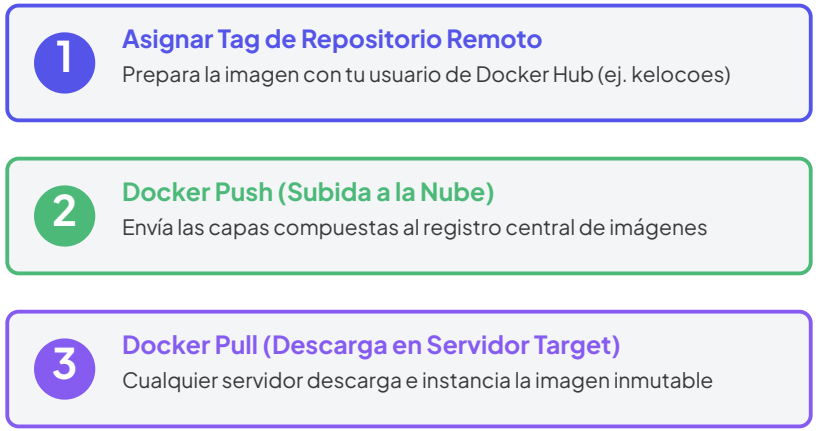
2. Subir (Push):

```
docker push kelocoes/springboot-app:v1
```

3. Descargar (Pull):

```
docker pull kelocoes/springboot-app:v1
```

Diagrama del Flujo de Publicación



Universidad ICESI

Docker Compose

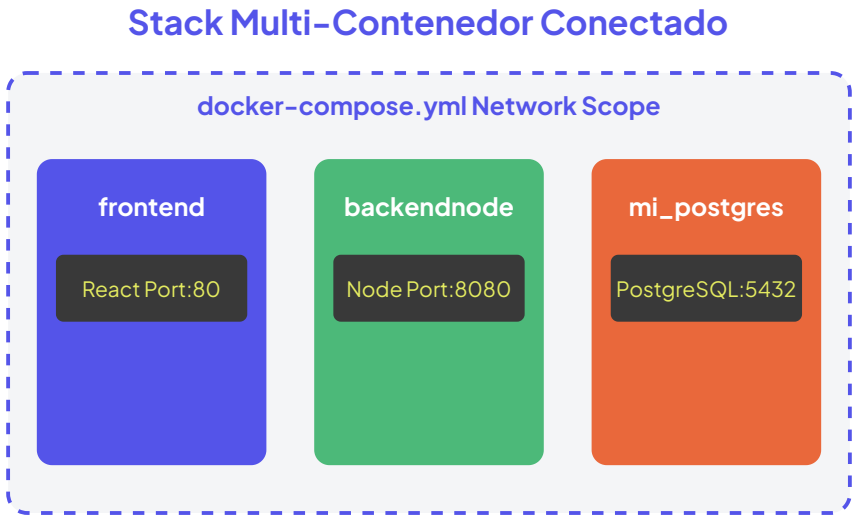
Orquestación Multi-Contenedor

¿Cómo administramos aplicaciones compuestas por Frontend, Backend API y Base de Datos sin lanzar comandos individuales?

¿Qué es Docker Compose y para qué sirve?

Docker Compose es una herramienta oficial para definir y ejecutar aplicaciones Docker compuestas por **múltiples contenedores**.

- **Archivo Declarativo:** Utiliza un archivo en formato YAML (`docker-compose.yml`) para configurar los servicios, redes y volúmenes.
- **Un Solo Comando:** Levanta o destruye toda la arquitectura completa con `docker-compose up` o `down`.
- **Redes Automáticas:** Crea automáticamente una red interna donde los servicios se comunican usando su propio nombre de servicio como *hostname*.



Universidad ICESI

Antes y Después: Script Shell vs Compose

Antes (Script manual .sh)

```
# Crear red manualmente
docker network create mi_red

# Levantar base de datos
docker run -d --name db --network mi_red \
 -e POSTGRES_PASSWORD=sec \
 postgres:15-alpine

# Compilar y levantar backend
docker build -t mi-api ./backend
docker run -d --name api --network mi_red \
 -p 8080:8080 mi-api

# Difícil de mantener y monitorear
```

Después (docker-compose.yml)

```
version: '3.8'
services:
 backendnode:
  build: ./backend
  ports: ["8080:8080"]
  environment:
   - DB_HOST=mi_postgres
 mi_postgres:
  image: postgres:15-alpine
  environment:
   - POSTGRES_PASSWORD=sec
```

Comandos Principales de Docker Compose

Comando Compose	Descripción y Caso de Uso
`docker-compose up`	Inicia todos los servicios definidos en el archivo `docker-compose.yml`. Acompáñalo de `-d` para ejecutarlo en segundo plano.
`docker-compose up --build`	Reconstruye las imágenes de los servicios locales y arranca la aplicación actualizada.
`docker-compose down`	Detiene y elimina limpiamente los contenedores, redes y volúmenes creados por el comando `up`.
`docker-compose logs -f`	Muestra y sigue la consola unificada de registros (logs) de todos los servicios del stack simultáneamente.
`docker-compose exec <servicio> <cmd>`	Ejecuta un comando interno dentro del contenedor en ejecución de un servicio específico (ej. `docker-compose exec backendnode sh`).