Universidad Intercontinental de la Empresa

Asignatura

Arquitectura del Software 

Profesor

Yago Fontenla Seco

{*yago.fontenla1@uie.edu*}

Virtualización

La **virtualización** es la creación de una versión virtual de un sistema operativo, servidor, dispositivo de almacenamiento o red que permite ejecutar **múltiples entornos aislados** sobre el mismo hardware físico.

En otras palabras, una misma máquina física puede comportarse como varias máquinas independientes.

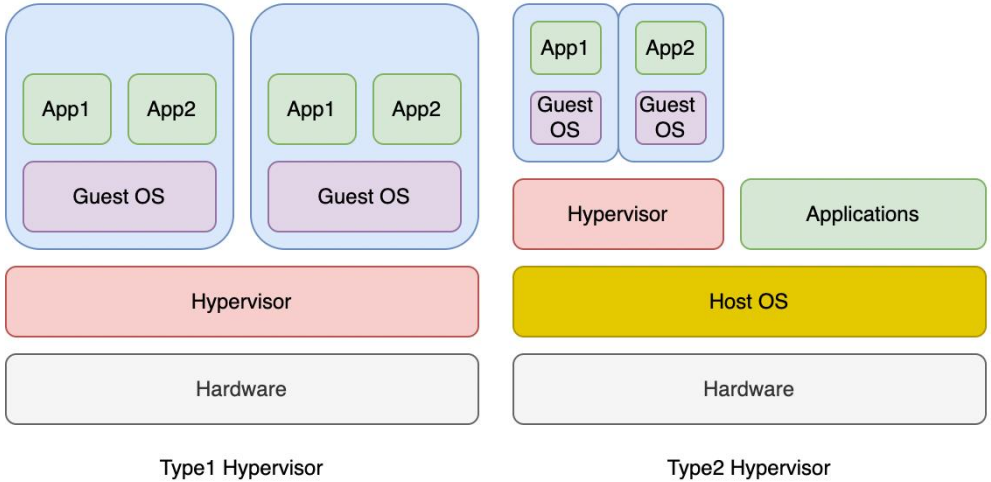
- **Década de 1960:** IBM desarrolla los primeros *mainframes virtualizados* (IBM CP-40, CP-67).
- **Década de 1990:** resurgen las técnicas de virtualización en entornos x86 gracias al abaratamiento del hardware y la necesidad de consolidar servidores.
- **2000–2010:** la virtualización se convierte en el pilar de los **centros de datos** y **nubes privadas**, permitiendo aprovechar mejor los recursos físicos.
- **Actualidad:** la virtualización es la base de **nubes públicas (AWS, Azure, GCP)**

Virtualización

Un hipervisor (o monitor de máquinas virtuales) es el software que permite crear y gestionar máquinas virtuales (VMs).

- **Tipo 1 (bare-metal):** corre directamente sobre el hardware (ej. VMware ESXi, Hyper-V, Xen).

- **Tipo 2 (hosted):** se ejecuta sobre un sistema operativo anfitrión (ej. VirtualBox, VMware Workstation).



Contenerización

La **contenerización** consiste en empaquetar una aplicación junto con todas sus dependencias (librerías, binarios, configuraciones) dentro de una **unidad ligera y aislada llamada contenedor**.

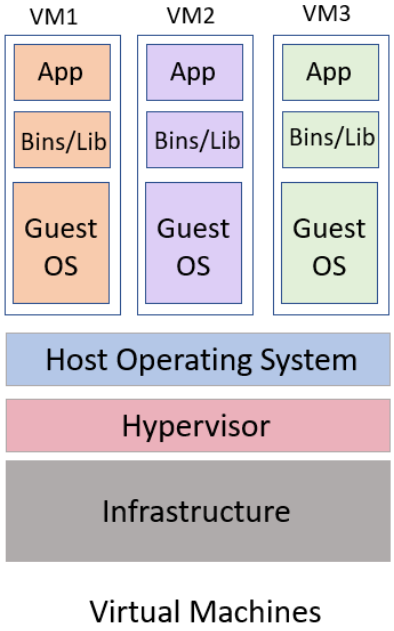
- **Años 2000:** surgen mecanismos de aislamiento a nivel de sistema operativo (como chroot, LXC en Linux).
- **2013:** nace Docker, que simplifica y estandariza la creación y distribución de contenedores.
- **2014–2016:** adopción masiva en entornos DevOps y despliegues en la nube.

Contenerización vs Virtualización

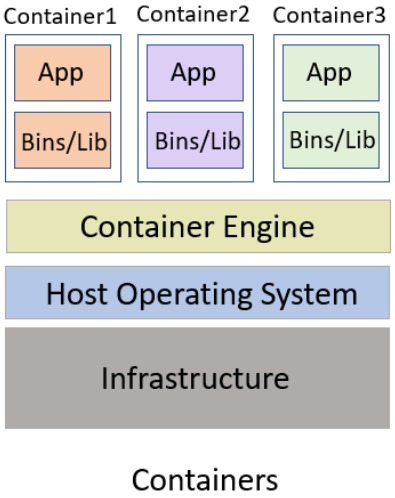
A diferencia de las VMs, **los contenedores comparten el kernel** del sistema operativo anfitrión.

Se ejecutan de forma aislada en espacio de usuario, gracias a tecnologías como namespaces y cgroups de Linux.

Cada **contenedor** contiene solo lo necesario para correr su aplicación.



Virtual Machines



Containers

Docker

Docker es una plataforma que permite **empaquetar, distribuir y ejecutar** aplicaciones dentro de contenedores.

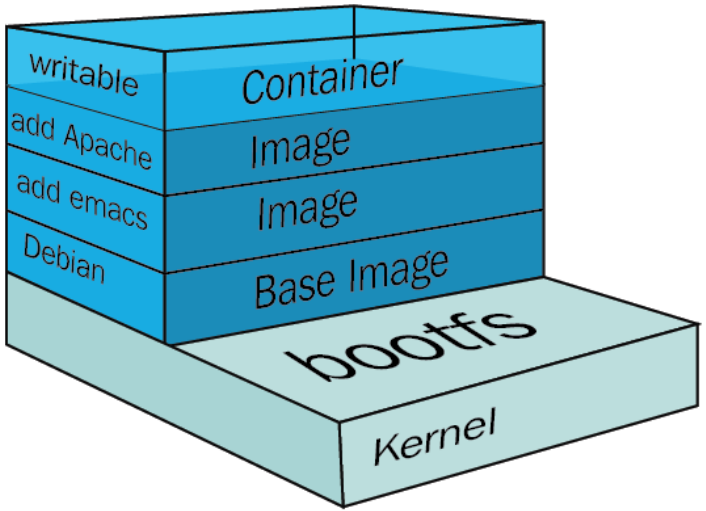
Componentes principales:

- **Docker Engine:** motor que ejecuta y gestiona contenedores.
- **Docker CLI:** interfaz de línea de comandos.
- **Docker Hub:** repositorio público de imágenes.

Conceptos clave:

- Una **imagen Docker** es una plantilla inmutable.
  Un **contenedor** es una instancia en ejecución de esa imagen.

Docker



Kernel: Núcleo del sistema operativo anfitrión. Todos los contenedores lo comparten. Gestiona recursos de hardware (CPU, memoria, red, almacenamiento).

bootfs (Boot File System): Sistema de archivos que contiene los archivos necesarios para arrancar el entorno base.

Base Image: Imagen inicial sobre la cual se construyen las demás capas. Suele ser una distribución ligera de Linux (como Debian, Alpine, Ubuntu).

Image Layers: Cada instrucción del Dockerfile crea una **nueva capa**. Las capas son **de solo lectura** y se apilan sobre la base. Se reutilizan gracias a la **caché de Docker**, acelerando la construcción de imágenes.

Container: Capa superior de **lectura y escritura** creada al ejecutar un contenedor. Guarda los cambios hechos durante la ejecución (logs, archivos temporales, configuraciones).

Dockerfile

Dockerfile: archivo de texto que define **cómo construir una imagen**.

```
# 1. Imagen base
FROM python:3.10

# 2. Directorio de trabajo
WORKDIR /app

# 3. Copiar archivos del proyecto
COPY . .

# 4. Instalar dependencias
RUN pip install -r requirements.txt

# 5. Exponer el puerto de la aplicación
EXPOSE 5000

# 6. Comando de arranque
CMD ["python", "app.py"]

  return go(f, seed, [])
}
```

- **FROM:** indica la imagen base sobre la que se construye la nueva.
- **WORKDIR:** define el directorio donde se ejecutarán los comandos.
- **COPY:** copia archivos locales al contenedor.
- **RUN:** ejecuta comandos en tiempo de construcción.
- **EXPOSE:** documenta el puerto que usará el contenedor.
- **CMD:** indica el comando que se ejecutará al iniciar el contenedor.

Crear una imagen

```
docker build -t miapp .
```

Usa el Dockerfile del directorio actual (.) para construir una imagen.
La opción -t miapp asigna una etiqueta (tag) a la imagen, facilitando su identificación.

Proceso:

- Docker ejecuta cada instrucción del Dockerfile y crea una capa inmutable por cada paso.
- Al finalizar, genera una imagen que contiene todo lo necesario para ejecutar la aplicación.

Ejecutar un contenedor

```
docker run -p 5000:5000 miapp
```

Crea y ejecuta un contenedor a partir de la imagen miapp.
La opción -p 5000:5000 mapea el puerto 5000 del contenedor al puerto 5000 del host, permitiendo el acceso desde el exterior.

Proceso:

- Docker añade una **capa de escritura** encima de la imagen (para logs y cambios).
- La aplicación se ejecuta dentro de un entorno aislado pero compartiendo el kernel del host.

Compartir la imagen

```
docker push usuario/miapp
```

Sube la imagen al repositorio público (o privado) de Docker Hub.
Otros desarrolladores o servidores pueden **descargarla con docker pull**.

Proceso:

- Docker divide la imagen en capas y solo sube las que no existen ya en el repositorio.
- Requiere haber hecho docker login previamente con tus credenciales.

Ciclo de vida de una aplicación contenerizada

1. Escribir el Dockerfile.
2. Crear la imagen: docker build -t miapp .
3. Ejecutar un contenedor: docker run -p 5000:5000 miapp
4. Compartir la imagen en Docker Hub: docker push usuario/miapp
5. Desplegarla en producción, CI/CD o Kubernetes.

Conclusiones

La **contenerización** representa una evolución clave en la gestión del software moderno: permite **ejecutar, escalar y desplegar aplicaciones** de forma más ágil, consistente y eficiente.

Docker se ha consolidado como el **estándar de facto** para la creación y ejecución de contenedores, facilitando la adopción de arquitecturas **cloud-native** y **basadas en microservicios.**

Comprender el uso del **Dockerfile** y el flujo **build → run → deploy** es fundamental para todo **arquitecto de software o ingeniero DevOps**, ya que garantiza la portabilidad, reproducibilidad y automatización en los procesos de desarrollo y despliegue.

Universidad Intercontinental de la Empresa

Asignatura

Arquitectura del Software 

Profesor

Yago Fontenla Seco

{*yago.fontenla1@uie.edu*}