Virtualización

TB022 - Esteban - Riesgo - Kristal

Qué es

Creación de ambientes computacionales virtuales, en lugar de físicos, a partir de un sistema de hardware real.

El ambiente virtual creado, simula ser un ambiente nativo específico con el cual el software puede interactuar sin preocuparse.

El manejador del ambiente virtual corre sobre un ambiente real y se encarga de traducir las instrucciones del software virtualizado para que el ambiente real y el hardware lo entiendan.

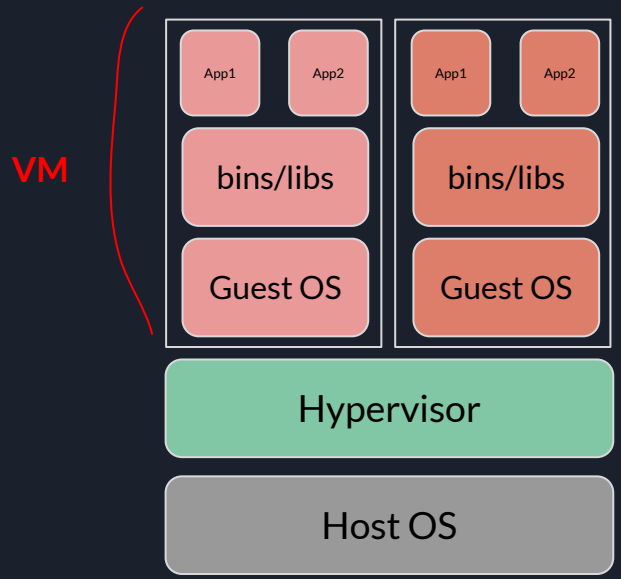
Se dice que hay "una capa extra" en el procesamiento con respecto al desarrollo tradicional.

Tiene el beneficio de que el ambiente virtualizado es replicable para cualquier persona (developer) que lo quiera usar, haciéndolo totalmente homogéneo. Por otro lado, al tener estas capas extra de traducción, el poder de procesamiento se ve reducido.

Cuando se hable de un ambiente virtual se le referirá como *VM* o *Virtual Machine*.

Este recurso no es algo nuevo, desde los 60s que se utilizan VMs de una forma u otra.

Estructura de virtualización



El SO host corre nativamente sobre el hardware.

El hypervisor es un programa como cualquier otro que corre sobre el SO nativo. Es quien se encarga de manejar el ambiente virtual: gestionar sus recursos y enviarle al host OS las instrucciones para el hardware. Además maneja los métodos de entrada y salida (periféricos, pantalla) que el ambiente virtual puede usar.

Dentro del hypervisor, puede haber multitud de SO huéspedes corriendo. Cada uno se inicia y corre de manera independiente y ajenos a que están siendo virtualizados. Es indistinto para ellos.

Dentro de cada guest OS, se inicializan sus librerias y dependencias normalmente, y se corren las aplicaciones que se deseen.

Ventajas de la virtualización

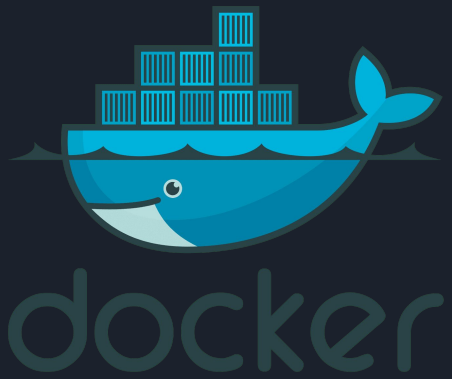
Agilidad: levantar VMs es relativamente sencillo, pues sólo se debe configurar una vez, luego simplemente se replica. Es muy útil para replicar ambientes productivos o hacer set ups comunes para distintos escenarios.

Escalabilidad: es más sencillo montar la aplicación en servidores. Aumentar la capacidad de procesamiento agregando servidores es trivial.

Portabilidad: no importa el host OS, la VM es la misma para todos, y puede ser usada en cualquier ambiente. Simplemente hay que utilizar el hypervisor adecuado según el host OS.

Seguridad: el hypervisor se encarga de manejar las conexiones a las VM. Se puede especificar exactamente qué se quiere permitir y exponer para evitar posibles ataques imprevistos.

Docker



Qué es

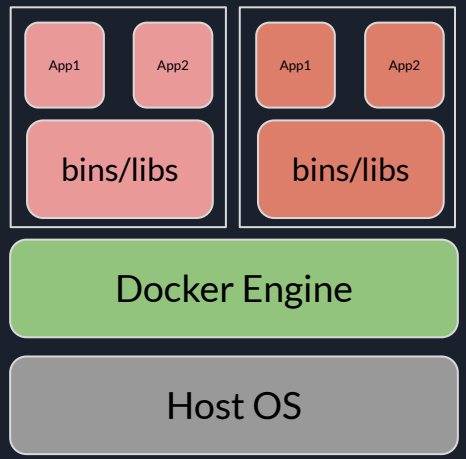
Docker es una aplicación que propone una manera distinta de virtualizar un ambiente de manera más eficiente.

Es relativamente reciente, su primer release fue en el 2013 pero hoy en día es el sistema de virtualización más usado en el mundo. Prácticamente todas las aplicaciones productivas lo usan de una manera u otra.

Está orientada al desarrollo, mientras que las VMs originales eran de uso general. Debido a esto, provee varias facilidades para los desarrolladores: repositorio centralizado de imágenes, manejo mediante CLI, es open source.

Su principal diferencia con las VMs tradicionales es que no corren un guest OS, ahorrándose una capa.

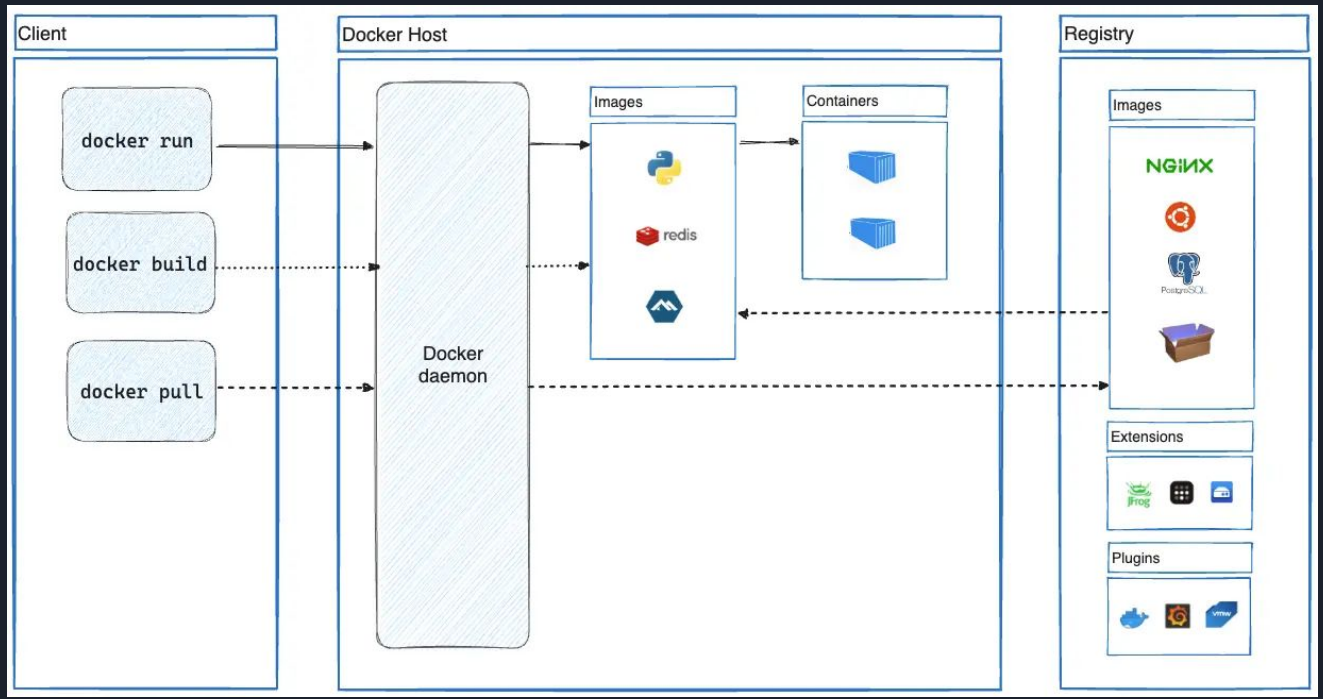
Capas en Docker



Docker no maneja sistemas operativos sino que maneja imágenes. Estas son compilados de programas y librerías que utiliza como base para correr aplicaciones.

Estas imágenes se *buildean* para generar *containers* donde correrán las aplicaciones de manera aislada.

Arquitectura



Definiciones

Cuales son los componentes en juego

Cliente (docker)

Aplicación CLI que permite al usuario interactuar con Docker.

Docker Host

Sistema Operativo del host (host OS) con el cual interactúa el usuario normalmente.

Docker Daemon (dockerd)

Recibe los comandos del cliente y maneja las imágenes, containers y todo lo demás

Docker registry

Repositorio donde se almacenan imágenes. Puede ser público o privado. Docker Hub es el registro público donde se almacanenan la mayoría de las imágenes disponibles para todo el mundo.

Docker Hub

- Docker Hub es un *registry* de imágenes. Es un lugar donde uno puede subir imágenes ya buildeadas para poder ser usadas directamente.
- Acá vamos a encontrar imágenes de servicios comunes, por ejemplo:
  - MySQL, Postgres, MongoDB (bases de datos)
  - Redis (Caché)
  - Apache, nginx (Servidores HTTP)
  - Python, Java, Go, etc. para correr aplicaciones sobre estos lenguajes
  - etc.

Definiciones

Qué objetos existen dentro del ecosistema

Imagen

Es un template de sólo lectura con instrucciones de cómo crear un *container*. Definen las dependencias necesarias para que la aplicación que se quiere virtualizar pueda correr. Estas dependencias van desde la base del sistema operativo que se quiere virtualizar, a cada una de las bibliotecas necesarias.

Container

Es una instancia ejecutable de una imagen. Una vez que un container es creado a partir de una imagen se puede ejecutar y el mismo correrá hasta que sea manualmente detenido. Al correr un container se le pueden agregar argumentos extra para cambiar su comportamiento, más allá de las definiciones dadas por la imagen.

El usuario puede conectarse a un container y ejecutar comandos como si estuviera en una terminal local, visualizar archivos, procesos, etc. todo lo que podría hacer si estuviera corriéndolo nativamente.

Generación de imágenes: Dockerfile

Instrucciones sobre cómo empaquetar la aplicación. Siempre va a estar basado en una imagen ya existente.

El archivo se basa en definir los pasos necesarios para setupear la imagen con todo lo necesario, para ello tenemos los siguientes comandos:

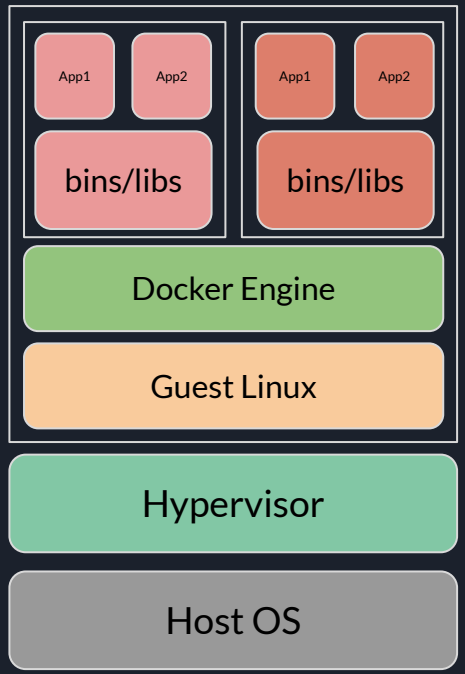
- `FROM <imagen>`: imagen base a utilizar con el OS
- `WORKDIR <path>`: el path en la imagen donde los archivos se van a copiar y donde se van a ejecutar los comandos
- `COPY <host-path> <image-path>`: copia los archivos del host a la imagen
- `RUN <comando>`: corre el comando especificado (bash)
- `ENV <nombre> <valor>`: setea variables de entorno
- `EXPOSE <puerto>`: puerto que queremos exponer en la imagen
- `USER <user>`: setea el user para las siguientes instrucciones
- `CMD ["<command>", "<arg1>", ...]` el comando final que se va a correr en la imagenluego de todos los pasos anteriores, junto con sus argumentos

Sobre el OS virtualizado

La virtualización con Docker vimos que se ahorra un paso al tener un Guest OS. Pero al construir una imagen necesitamos establecer una imagen base que tenga un OS. Entonces?

Esto que dijimos sólo es verdad si Docker se corre en sistemas Linux, ya que las distros utilizan el mismo Kernel base (mas o menos). Docker se aprovecha de esto y corre sobre el kernel del Host OS en vez de crear un nuevo desde cero. Más detalles acá y acá.

Si el Host OS es Windows, el stack de Docker es en realidad algo así:



Dockerfile: Fastapi

Para nuestra aplicación de FastApi, el Dockerfile deberia quedar mas o menos asi

```
FROM python:3.13

WORKDIR /app

# Instalamos las dependencias
COPY Pipfile Pipfile.lock ./
RUN python -m pip install --upgrade pip
RUN pip install pipenv && pipenv install --dev --system --deploy

# Copiamos el codigo fuente
COPY . .

EXPOSE 8000

# Usamos un user que no sea root (es opcional, puede ser cualquiera)
RUN useradd --create-home --shell /bin/bash app && \
   chown -R app:app /app
USER app

# Corremos las migraciones siempre
RUN pipenv run alembic upgrade head

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

Dockerignore

Como el .gitignore pero para ignorar archivos a copiar al container

```
__pycache__/
*.py[cod]
*$py.class
.pytest_cache/
.coverage
htmlcov/
alembic/__pycache__/
alembic/versions/__pycache__/
*.db
# Keep alembic.ini and migration files
!alembic/
!alembic/versions/
!alembic.ini
.DS_Store
.vscode/
.git
.gitignore
README.md
Dockerfile
.dockerignore
```

Probablemente falten agregar más archivos

Build

Ahora que el dockerfile está definido, hay que buildear la imagen

`docker build -t <nombre-imagen> <path>`

Construye la imagen a partir del Dockerfile del path especificado y le asigna un nombre de fantasía (para luego poder referenciarla más fácilmente, también tiene un ID único autogenerado).

En nuestro caso podemos poner algo asi:

`docker build -t fastapi-example .`

Podemos ver las imagenes creadas corriendo

`docker image ls`

Una vez terminados todos los pasos,
hay que correr el contenedor

Run

Ahora que se buildeó la imagen, hay que correr el contenedor

`docker run <nombre-imagen>`

Corre un container generado a partir de la imagen especificada.

En nuestro caso podemos poner algo asi:

```
docker run -p 8000:8000 fastapi-example .
```

Ante cualquier cambio en el código fuente, las dependencias o el Dockerfile mismo, debemos buildear nuevamente la imagen y luego correr con el container.

Run

Algunos argumentos útiles al correr un container

`-d`

Modo daemon. El container correrá en el background, no va a ocupar una pestaña de la terminal. Podemos seguir haciendo otras cosas en la misma.

Nos mostrará por pantalla el ID del container que se creó.

Podemos ver qué containers están corriendo con

```
docker ps
```

Podemos ver los logs de un container con

```
docker logs -f <container_id>
```

Y detenerlo con

```
docker stop <container_id>
```

Run

Algunos argumentos útiles al correr un container

`-it`

Modo interactivo con terminal. Se correrá el comando/script indicado en el container indicado y el output se mostrará por la terminal actual.
Ejemplo:

```
docker run -it <container> path/to/my/script.sh
```

También se puede usar para correr bash sobre el container y "abrir una consola" en el mismo.

```
docker run -it <container> bash
```

Esta operación puede ser muy útil para ver archivos de logs, interactuar con el sistema, ver que todos los archivos y cosas estén en el estado correcto.

Run

Algunos argumentos útiles al correr un container

Podemos ver los container detenidos con

```
docker ps -a
```

Y eliminar el que no se necesite con

```
docker rm <container_id>
```

o todos con

```
docker container prune
```

para auto eliminar el container una vez que se termina la ejecución debemos agregar `--rm` al comando `run`

Run all in one

Algunos argumentos útiles al correr un container

Para evitar llenar el registro de imágenes y containers que no queremos tener, podemos correr un único comando que nos permite buildear la imagen temporalmente, y correr un container que se auto limpiará luego de ejecutarse:

```
docker run --rm -it $(docker build -q .)
```

Docker Compose

Al trabajar en aplicaciones más complejas, en donde varios servicios interactúan, suele ser útil usar Docker Compose en lugar de usar Docker directamente.

Docker Compose nos permite declarar en un archivo YAML qué servicios necesitamos, qué configuración debe llevar cada uno, y levantarlos simultáneamente.

En nuestro caso, nos sería muy útil para poder tener corriendo nuestro BE y nuestro FE al mismo tiempo.

Dockerfile: Sveltekit

Este dockerfile debería funcionar para nuestro servidor de Sveltekit

```
FROM node:18-alpine

WORKDIR /app

RUN apk add --no-cache curl git

COPY package*.json ./

# Instalar dependencias
RUN npm ci --only=production=false

COPY . .

EXPOSE 5173

# Usamos un user que no sea root (es opcional, puede ser cualquiera)
RUN addgroup -g 1001 -S nodejs && \
   adduser -S sveltekit -u 1001 && \
   chown -R sveltekit:nodejs /app
USER sveltekit

CMD ["npm", "run", "dev", "--", "--host", "0.0.0.0", "--port", "5173"]
```

.dockerignore: Sveltekit

```
node_modules/
npm-debug.log*
.DS_Store
.vscode/
.idea/
*~
.git
.gitignore
README.md
Dockerfile
.dockerignore
docker-compose.yml
.svelte-kit/
build/
dist/
.output/
```

Docker Compose

En el directorio padre de nuestros proyectos vamos a crear un archivo docker-compose.yaml donde vamos a especificar los detalles de los servicios a correr

```yaml
version: '3.8'

services:
 fastapi_example:
   build:
     context: ./fastapi_example
     dockerfile: Dockerfile
   ports:
     - "8000:8000"
   environment:
     - PYTHONUNBUFFERED=1
   volumes:
     - ./fastapi_example:/app
   networks:
     - app_network
   restart: unless-stopped
```

qué versión de compose usar

la lista de servicios que van a ser corridos

cómo levantar cada servicio, con nombre y dockerfile

los puertos a exponer

variables de entorno a usar

sincroniza `host_path:container_path`

crea un red virtual en la cual todos los servicios que pertenezcan pueden comunicarse entre si por nombre

reinicia el container si el mismo crashea, pero no si es detenido manualmente

Docker Compose

En el directorio padre de nuestros proyectos vamos a crear un archivo docker-compose.yaml donde vamos a especificar los detalles de los servicios a correr

```yaml
 sveltekit_example:
   build:
     context: ./sveltekit_example
     dockerfile: Dockerfile
   ports:
     - "5173:5173"
   environment:
     - PUBLIC_API_URL=http://fastapi_example:8000
   volumes:
     - ./sveltekit_example:/app
     - /app/node_modules
   networks:
     - app_network
   restart: unless-stopped
   depends_on:
     - fastapi_example

networks:
 app_network:
   driver: bridge
```

indica qué servicios deben levantarse primero para poder empezar a levantar este

crea la red custom con este nombre, con el modo default

Docker Compose

<TODO> add comandos

Docker Desktop

Porque siempre hay una app además de la consola

Permite visualizar más fácilmente las imágenes y container creados y hacer operaciones con ellos.

Recomendamos siempre usar la consola y aprenderse los comandos y qué implica cada uno de ellos, pero ante operaciones complejas y multitud de imágenes y containers (sobretodo usando docker compose) es recomendable usar esta aplicación.

Fin