Universidad Francisco de Vitoria
UFV Madrid

Introducción a los contenedores

Programación II

Grado en Business Analytics

¿Por qué trabajar con entornos?

Universidad Francisco de Vitoria UFV Madrid

¿Por qué trabajar con entornos?

¿Qué pasa si tengo proyectos que usan diferentes versiones de módulos o paquetes para su funcionamiento?

===== Lo que me vale para el entorno de programación, me vale para un entorno de ejecución!!!=====

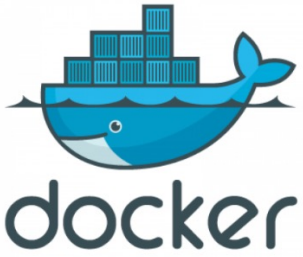
Universidad Francisco de Vitoria
UFV Madrid

Introducción a Docker

Universidad Francisco de Vitoria UFV Madrid

Introducción a Docker

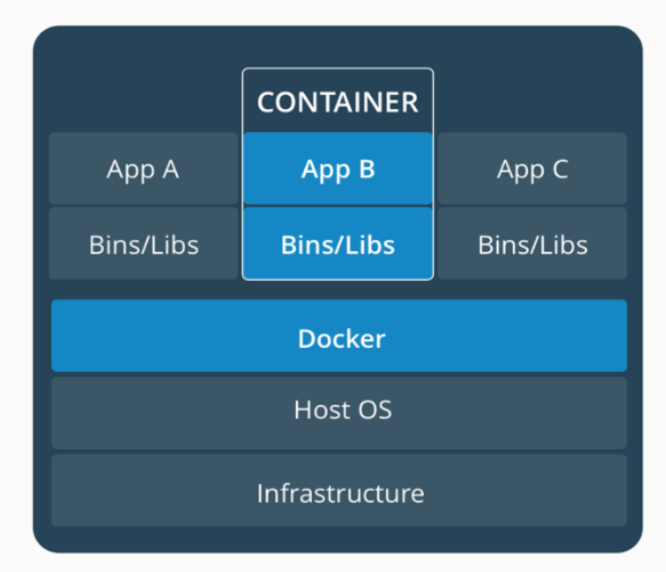
Lo que es



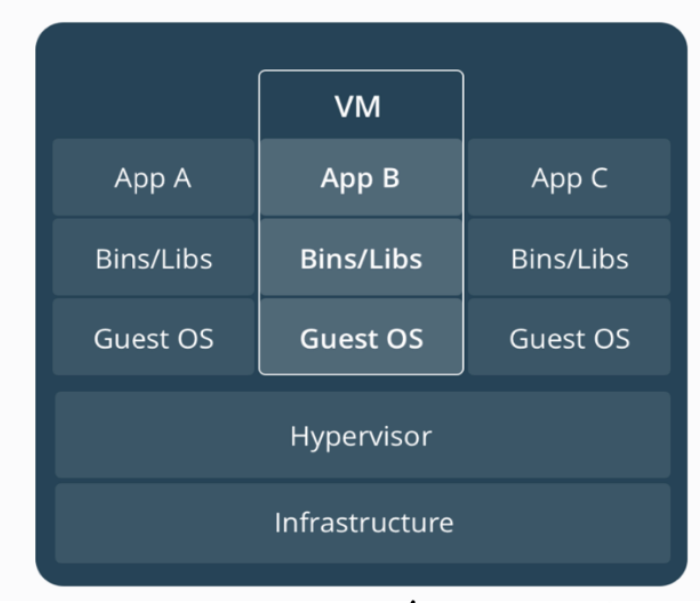
Docker es una tecnología que facilita el empaquetado y despliegue de aplicaciones: gestiona contenedores para el despliegue de aplicaciones, encapsulando el software que se va a ejecutar y sus dependencias

Universidad Francisco de Vitoria UFV Madrid

Introducción a Docker



- Los contenedores de Docker comparten el sistema operativo del host, encapsulando los procesos necesarios para ejecutar la aplicación (también de un sistema operativo, pero no necesariamente)
- En docker la gestión de recursos es dinámica y facilita la escalabilidad.
- Mucho más rápido y ligero

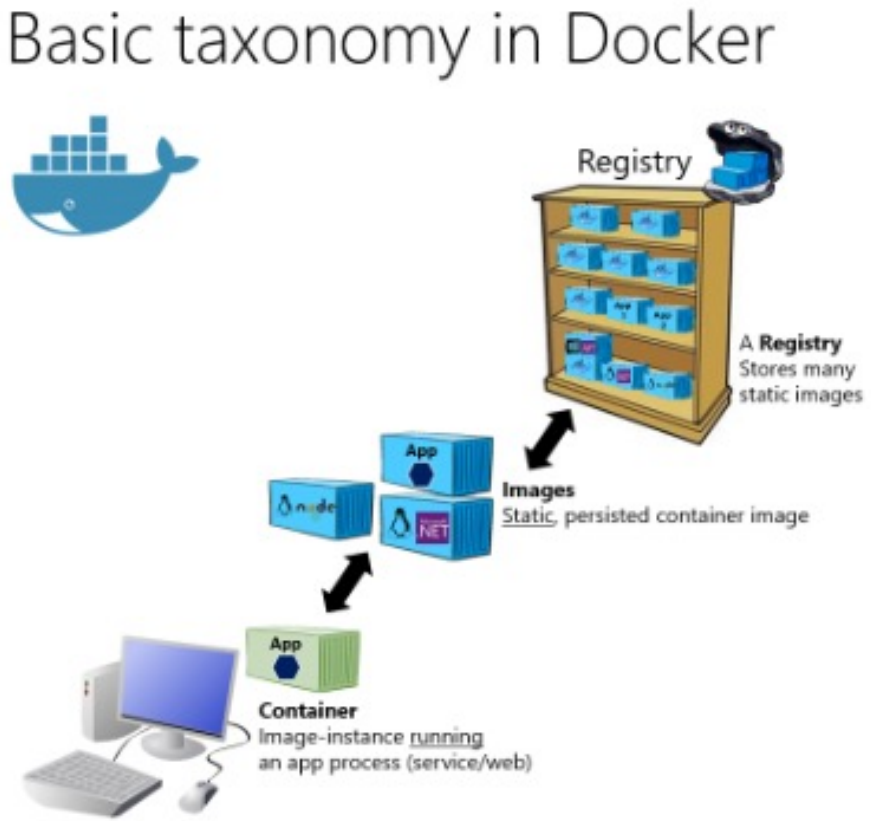


- Una máquina virtual es un ordenador virtualizado: se virtualiza todo el SSOO y el hypervisor genera capas de hw virtualizadas sobre la que se ejecuta la máquina virtual, esto es, se ejecuta un stack completo.
- Mucho más pesado y lento.

Principales conceptos

Universidad Francisco de Vitoria UFV Madrid

Principales conceptos



- **Hosted Docker Registry**
- **Docker Trusted Registry on-prem.**

→ **On-premises** ('n' private organizations)

- **Docker Hub Registry**
- Docker Trusted Registry on-cloud
- **Azure Container Registry**
- AWS Container Registry
- Google Container Registry
- Quay Registry
- Other Cloud

→ **Public Cloud** (specific vendors)

https://learn.microsoft.com/es-es/dotnet/architecture/microservices/container-docker-introduction/docker-containers-images-registries

Universidad Francisco de Vitoria UFV Madrid

Principales conceptos

Imagen:

Plantilla en la que se definen los componentes necesarios para ejecutar la aplicación. Esta plantilla se define en un fichero llamado ‘dockerfile’.

```
FROM Ubuntu:latest
RUN mkdir /tmp/offy
RUN Apt-get install
…
```

```
docker build…
docker run …
```

Contenedor:

Una imagen en ejecución que proporciona un entorno, aislado en cuanto a dependencias, y formado por todo lo necesario para su ejecución: software, librerías de sistemas, ajustes, etc.

Universidad Francisco de Vitoria UFV Madrid

Principales conceptos

Capas

Una imagen se basa en una imagen base que normalmente es la de un sistema operativo. A partir de esta imagen base se apilan diferentes paquetes y programas.

A cada una de las modificaciones que se realiza sobre la imagen mediante una instrucción en el dockerfile, se le llama capa.

Cuando se construye la imagen (docker build) se apila cada instrucción como una nueva capa y se mezclan en una única capa.

Universidad Francisco de Vitoria UFV Madrid

Principales conceptos

Volumen:

Mecanismo para almacenar los datos gestionados por un contenedor fuera del contenedor.

Pueden ser compartidos por varios contenedores y pueden trabajar sobre W o L.

```
$ docker volume create my-vol
$ ls -l /var/lib/docker/volumes/
$ Docker volume ls

$ docker run -v my-vol:/var/lib/mysql
mysql
```

https://docs.docker.com/storage/volumes/

Bind mounts:

Monta un directorio del sistema operativo en el contenedor. El rendimiento es menor que un volumen.

```
$ docker run --mount
type=bind,source=/data/mysql,target=/var/lib/
mysql mysql
```

tag:

Versión de una imagen

Dockerfile

Universidad Francisco de Vitoria UFV Madrid

Dockerfile

```
FROM godatadriven/pyspark

RUN apt-get install -y
netcat
ENTRYPOINT ["spark-
submit"]
EXPOSE 9998
```

Conjunto de instrucciones que indican a Docker cómo construir una imagen (es decir, generar un contenedor).

Universidad Francisco de Vitoria UFV Madrid

Dockerfile: principales instrucciones

- **FROM**: Identifica la imagen base sobre la que se basa el contenedor
- **RUN**: Ejecuta comandos del sistema operativo como una nueva capa para 'construir' (build) el contenedor.
- **CMD**: Comando por defecto cuando se ejecute el contenedor. Sólo puede haber una.
- **ENTRYPOINT**: Permite configurar el contenedor para que se ejecute como un ejecutable.
- **EXPOSE**: Informa a Docker de que el contenedor escuchará en los puertos identificados en tiempo de ejecución
- **ENV**: Especifica las variables de entorno: <key> <value>
- **ADD**: Copia nuevos archivos, directorios o fichero remotos desde una URL <src> y los añade al sistema de fichero de la imagen destino <path>
- **COPY**: Copia nuevos archivos o directorios desde <src> y los añade al sistema de ficheros del contenedor <dest>

Universidad Francisco de Vitoria UFV Madrid

Dockerfile: ejemplo 1

```
1   FROM tiangolo/uvicorn-gunicorn
2
3   RUN mkdir /fastapi
4
5   COPY requirements.txt /fastapi
6
7   WORKDIR /fastapi
8
9   RUN pip install -r requirements.txt
10
11  COPY . /fastapi
12
13  EXPOSE 8000
14
15  CMD ["uvicorn", "server:app", "--host", "0.0.0.0", "--port", "8000"]
```

Especifico la imagen de dockerhub que utilizo como base

Dentro del contenedor genero un directorio denominado /fastapi

De mi directorio local, copio requirements.txt a /fastapi (en el contenedor)

Establezco /fastapi como mi directorio de trabajo – cd /fastapi

Instalo los paquetes

Copio el resto de ficheros a /fastapi

Abro puerto 8000

Ejecuto uvicorn: equivalente a `uvicorn server:app –host 0.0.0. –port 8000`

Universidad Francisco de Vitoria UFV Madrid

Dockerfile: ejemplo 2

```
FROM godatadriven/pyspark

RUN apt-get install -y netcat
ENTRYPOINT ["spark-submit"]
EXPOSE 9998
```

Fijaos, en este caso he especificado un ENTRYPOINT, es decir, le digo que al contenedor se le puede llamar como un ejecutable y que su comando es spark-submit.

Cuando hago un docker-run, le pasaré los comandos que necesita spark-submit.

```
docker run -v $(pwd):/job sparkyimg  \
    --name "ejemplos" \
    --master "local[*]" \
    --conf
"spark.ui.showConsoleProgress=True" \
    --conf "spark.ui.enabled=False" \
    /job/spark_streaming_ej.py
```

Comandos

Universidad Francisco de Vitoria UFV Madrid

Principales comandos

1. Vemos si existen imágenes o contenedores ejecutándose

$ **docker images**: lista todas las imágenes que tienes descargadas o has definido.

$ **docker ps**: lista todos los contenedores en ejecución
$ **docker ps -a**: lista todos los contenedores, estén en ejecución o no
$ **docker ps -s**: lista todos los contenedores en ejecución especificando su tamaño , estén en ejecución o no

Universidad Francisco de Vitoria UFV Madrid

Principales comandos

2. Generamos un contenedor a partir de un Dockerfile

```
$ docker build -t micontenedor .:
```

Genera un contenedor a partir de la imagen especificada en un dockerfile. El contenedor se denomina 'micontenedor'.

El Dockerfile que se encuentra en el directorio actual (.). Si el dockerfile se denominase de otra forma, habría que sustituir . por el nombre del archivo

Universidad Francisco de Vitoria UFV Madrid

Principales comandos

3. Ejecutamos un contenedor

$ **docker run ubuntu**: ejecuta la imagen denominada ubuntu

$ **docker run -i ubuntu**: ejecuta en modo interactivo la imagen denominada ubuntu

$ **docker run ubuntu sleep 5**: ejecuta la imagen Ubuntu, sobreescribiendo el comando por defecto. En este caso, el comando pasaría a ser sleep

$ **docker run -p puerto_host:puerto_contenedor ubuntu**

$ **docker run -v host:ruta_contenedor <image-name>**: En este caso, 'host' es un volumen docker o un directorio del sistema operativo host.

Universidad Francisco de Vitoria UFV Madrid

Principales comandos

4. Ejecutamos o paramos un contenedor que ya ha sido construido y que hemos corrido en alguna ocasión

```
$ docker start ubuntu: 'enciende' el contenedor ubuntu

$ docker stop ubuntu: 'para' el contenedor Ubuntu
```

Universidad Francisco de Vitoria UFV Madrid

Principales comandos

5. Ejecutamos un comando específico dentro del contenedor.

$ **docker exec -it ubuntu <comando>**: ejecuta <comando> en el contenedor ubuntu, en modo interactivo (i) y con un pseudo terminal (t). Un uso muy típico es el de utilizar este comando para entrar en el terminal de un contenedor en ejecución: docker exec -it ubuntu /bin/bash

Otros comandos:

$ **docker kill contenedor**
$ **docker rmi imagen**
$ **docker rm contenedor**

Orquestación con docker compose

Orquestación con docker compose

Docker compose facilita la ejecución de varios contenedores compartiendo una misma red y otros recursos.

Orquestación con docker compose

yaml

```yaml
version: '3'

services:

  fastapi:
    build: fastapi/
    ports:
      - 8000:8000
    networks:
      - deploy_network
    container_name: fastapi
    volumes:
      - mis_datos:/var/lib/docker/volumes/aa-streamlit-fastapi_mis_datos/_data
  streamlit:
    build: streamlit/
    depends_on:
      - fastapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlit
    volumes:
      - mis_datos:/var/lib/docker/volumes/aa-streamlit-fastapi_mis_datos/_data
networks:
  deploy_network:
    driver: bridge
```

Del mismo modo que docker tiene su dockerfile, docker compose tiene un fichero en formato yaml (compose.yaml) donde se definen los servicios (contenerdore) y los recursos (redes, volúmenes o bases de datos)

Orquestación con docker compose

yaml

```yaml
version: '3'
services:
  fastapi:
    build: fastapi/
    ports:
      - 8000:8000
    networks:
      - deploy_network
    container_name: fastapi
    volumes:
      - mis_datos:/var/lib/docker/volumes/aa-streamlit-fastapi_mis_datos/_data
  streamlit:
    build: streamlit/
    depends_on:
      - fastapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlit
    volumes:
      - mis_datos:/var/lib/docker/volumes/aa-streamlit-fastapi_mis_datos/_data
networks:
  deploy_network:
    driver: bridge
```

`services` es la única sección obligatoria, y es donde vamos a especificar los servicios (contenerdores) que se van a levantar y a ejecutar.

En este caso se han definido dos: `fastapi` y `streamlit`

Orquestación con docker compose

yaml

```
version: '3'
services:
  fastapi:
    build: fastapi/
    ports:
      - 8000:8000
    networks:
      - deploy_network
    container_name: fastapi
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
  streamlit:
    build: streamlit/
    depends_on:
      - fastapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlit
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
networks:
  deploy_network:
    driver: bridge
```

`build` define directorio donde está el dockerfile donde se va a realizar un `docker build`.

Se puede especificar una o varias dependencias. Al establecer dependencias, estamos diciendo que para levantarse un servicio, el otro tiene que estar levantado.

Orquestación con docker compose

yaml

```
version: '3'
services:
  fastapi:
    build: fastapi/
    ports:
      - 8000:8000
    networks:
      - deploy_network
    container_name: fastapi
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
  streamlit:
    build: streamlit/
    depends_on:
      - fastapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlit
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
networks:
  deploy_network:
    driver: bridge
```

También podríamos haber utilizado una imagen de un registro docker. Por ejemplo, podríamos haber descargado la imagen de mariadb y haberla definido así:

```
mariadb:
        image: 'mariadb:latest'
```

Orquestación con docker compose

yaml

```
version: '3'
services:
  fastapi:
    build: fastapi/
    ports:
      - 8000:8000
    networks:
      - deploy_network
    container_name: fastapi
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
  streamlit:
    build: streamlit/
    depends_on:
      - fastapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlit
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
networks:
  deploy_network:
    driver: bridge
```

`ports` puertos del contenedor que se van a exponer

Orquestación con docker compose

yaml

```
version: '3'
services:
  fastapi:
    build: fastapi/
    ports:
      - 8000:8000
    networks:
      - deploy_network
    container_name: fastapi
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
  streamlit:
    build: streamlit/
    depends_on:
      - fastapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlit
    volumes:
      - mis_datos:/var/lib/docker/volumes/<midirectorio>
networks:
  deploy_network:
    driver: bridge
```

`networks` son capas que permiten conectar los servicios.
En cada servicio, tenemos que especificar qué red va a utilizar.
En la sección de network, especificamos las características de esa conexión.
Una red de tipo bridge, crea una conexión privada con resolución automática DNS para nombres del host de los contenedores, si los hubiera.

Orquestación con docker compose

Yaml: otros parámetros útiles

```
Env: permite definir variables de entorno
Config: puedes especificar ficheros de configuración para tus servicios,
de forma que no tengas que reiniciar y levantar el contenedor para
modificarla.
Secrets: gestión de credenciales y contraseñas
```

Universidad Francisco de Vitoria UFV Madrid

Orquestación con docker compose

Comandos

$ **docker compose up/down .:**

Levanta o cierra los servicios especificados en el docker compose

$ **docker compose ps:** muestra los contenedores corriendo

$ **docker compose logs:** muestra los logs

Universidad Francisco de Vitoria
UFV Madrid

Introducción a los contenedores

Programación II

Grado en Business Analytics