Apuntes: Semana 9 Clase 1

Fecha: 20/09/2022
Autor: Deyan Sanabria Fallas #2021046131

Funcionamiento de Docker

Docker tienen sus propios contenedores donde se aislan del sistema operativo y no dependemos de ello.

¿Como funciona una aplicación sin Docker?

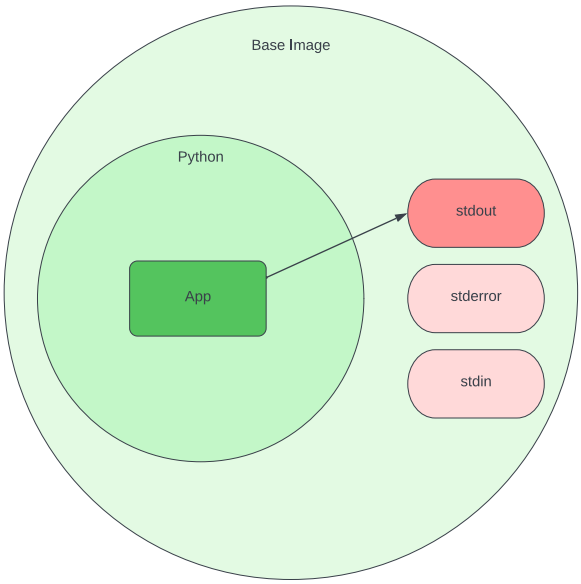
Desarrollar un programa en algún lenguaje como Python puede causar la necesidad de instalar librerías. Estas librerías pueden tener conflictos si por ejemplo tenemos dos aplicaciones, una en Python 3 y otra en Python 2 que requieren mismas librerías, pero para sus versiones de Python respectivas. Adicionalmente, al pasar estos programas a otro sistema operativo u otra computadora con el mismo sistema, estamos atenidos a que estos contengan las librerías que usan nuestro programa.

¿Como funciona una aplicación con Docker?

Docker nos permite crear contenedores, estos contenedores no dependen del sistema operativo donde Docker se esté ejecutando, en cambio le podemos especificar que librerías se necesitan para nuestras aplicaciones y Docker automáticamente creara un ambiente con la aplicación donde tenga todo lo que necesite para su ejecución correcta y además los contenedores se aíslan unos de otros por lo que podríamos tener una aplicación de Python 2 en un contenedor y otra de Python 3 en otro contenedor que funcionen simultáneamente pero no tengan conflictos. Al no depender del sistema operativo donde se ejecute Docker, podríamos instalar la misma imagen con el contenedor de las aplicaciones de Python en otro sistema como Linux y seguiría funcionando de manera correcta.

¿Como funcionan los contenedores?

Podriamos decir que los contenedores se dividen por capas.



Bases Images

Suelen venir de un sistema operativo, a continuación algunas familias de linux:

- **Debian**
  - Debian
  - Ubuntu
  - Mandriva
- **Fedora**
  - *Fedora*
  - *CentOS*
  - *RedHat*
- **Alphine**

Sesiones de Linux

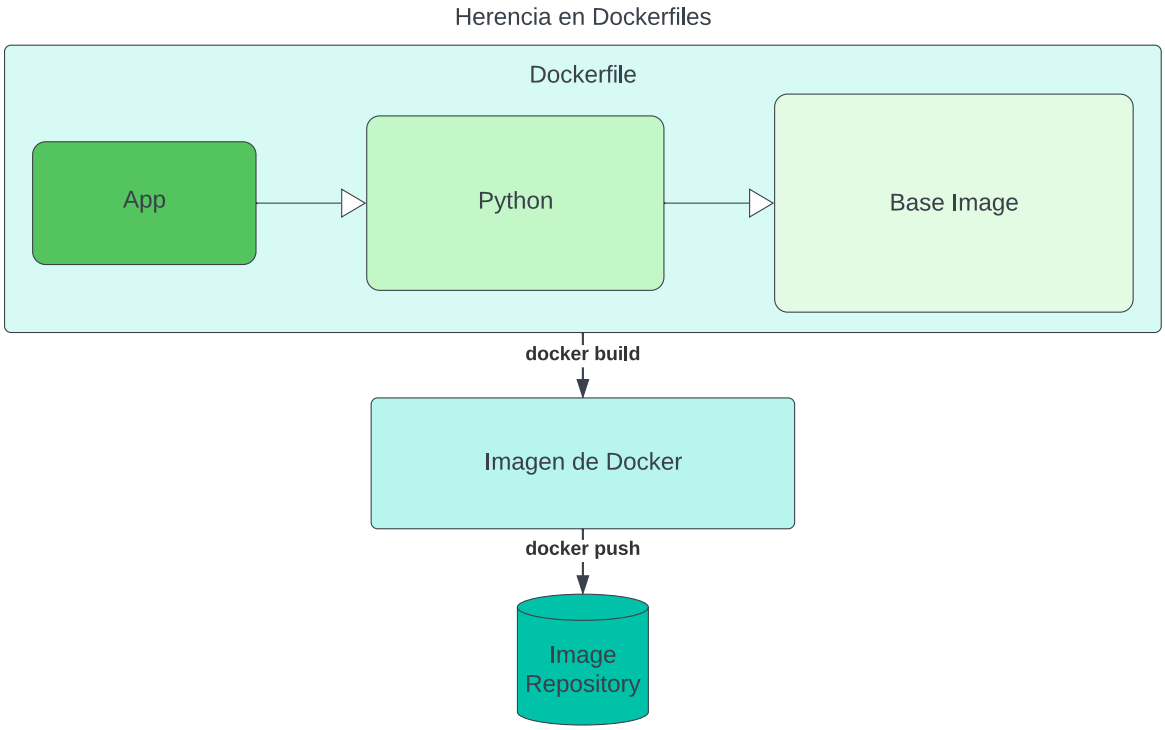
El ciclo de vida de una imagen de Docker es el tiempo de vida de una aplicación. Linux suele tener sessiones las cuales tienen lo siguiente:

- stdout: Las aplicaciones se conectan al stdout. La imagen de Docker se apodera del stdout y mientras la app este viva, el contenedor tambien.
- stderror
- stdin

Dockerfile para la creación de Imagenes de Docker

Un dockerfile nos dice como construir una imagen

- **docker build**: Sigue los pasos del dockerfile para construir la imagen
- **docker push**: Sube la imagen al repositorio de docker



Creación de Dockerfile

Codigo de ejemplo:

```
FROM python:3

WORKDIR /usr/src/app

COPY requirements.txt ./ RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD [ "python", "./your-daemon-or-script.py" ]
```

Explicacion del codigo

- *FROM python:3*
  -> '*python*' sería la imagen que queremos usar.
  -> '*3*' sería la versión de la imagen que queremos usar. (para el ejemplo del profesor usa la '*3.10.7-slim-bullseye*')
- '*WORKDIR /usr/src/app*'
  -> '*/usr/src/app*' esta sería la dirección de donde trabajemos o en otras palabras donde pondremos nuestra aplicación.
- '*COPY requirements.txt ./*'
  -> '*requirements.txt*' este documento de texto (si lo colocamos, o sea es opcional), es donde tendremos las librerías que se van a instalar, en esta instrucción se copia dentro del directorio de trabajo, cada línea del documento es una librería.
- '*RUN pip install --no-cache-dir -r requirements.txt*'
  -> '*pip install --no-cache-dir -r requirements.txt*'' este es el comando que ejecutara la instrucción '*RUN*', en este caso es un comando simple que le indica al *'Package Installer for Python'* que instale las librerías ubicadas en el documento de *'requerimientos.txt'*
- '*COPY . .*'
  -> '*. (el primero)*' esta es la ubicación donde estan los archivos de nuestro programa.
  -> '*. (el segundo)*' esta es la ubicación donde se copiaran los archivos de nuestro programa.
- '*CMD ["python", "./your-daemon-or-script.py"]*'
  -> '*["python", "./your-daemon-or-script.py"]*' esto sería un arreglo donde se pondrá el comando que ejecutara el contenedor para iniciar el programa, cada elemento del arreglo sería el cómo se colocara el comando separado por espacios.

Tips e infomación importante

- Usar version '*slim*'.
- '*rc*' suele significar '*release candidate*'.
- '*alphine*' es muy liviano, para pruebas puede que haya que instalar algunas aplicaciones porque no viene por defecto.
- El comando '*docker images*' muestra todas las imagenes que tenemos disponibles en nuestro Docker.
- '*-u*' es util poner el '*-u*' como argumento en el comando que ejecutara el programa de python (ej: '*CMD ["python", "-u", "./your-daemon-or-script.py"]*'), esto con el objetivo de que se muestren los prints al momento que se mandan y no cuando muera el contenedor.
- El comando '*docker -v run <imagen>*' muestra todas las imagenes que tenemos disponibles en nuestro Docker.
- El comando '*docker ps*' muestra los contenedores activos y sus nombres.
- El comando '*docker container exec -it <container> /bin/bash*' nos permite crear una sesion de bash en el contenedor para ejecutar comandos.
- Usar '*environmental variables*' para definirlas en el Heml Chart e importarlas en python
- Revisar cual es el usuario de '*RabbitMQ*' en '*values.yaml*' para conectar los programas de Python
- '*RabbitMQ*' tiene el problema que deja el Persistent Volumen cuando se desinstala y al reinstalar usa el mismo, pero cambia el secret de la contraseña, se recomienda borrar el Persistent Volumen cada vez que se desinstale

Compilado del Dockerfile

Comando

Se ejecuta dentro de la carpeta con el dockerfile

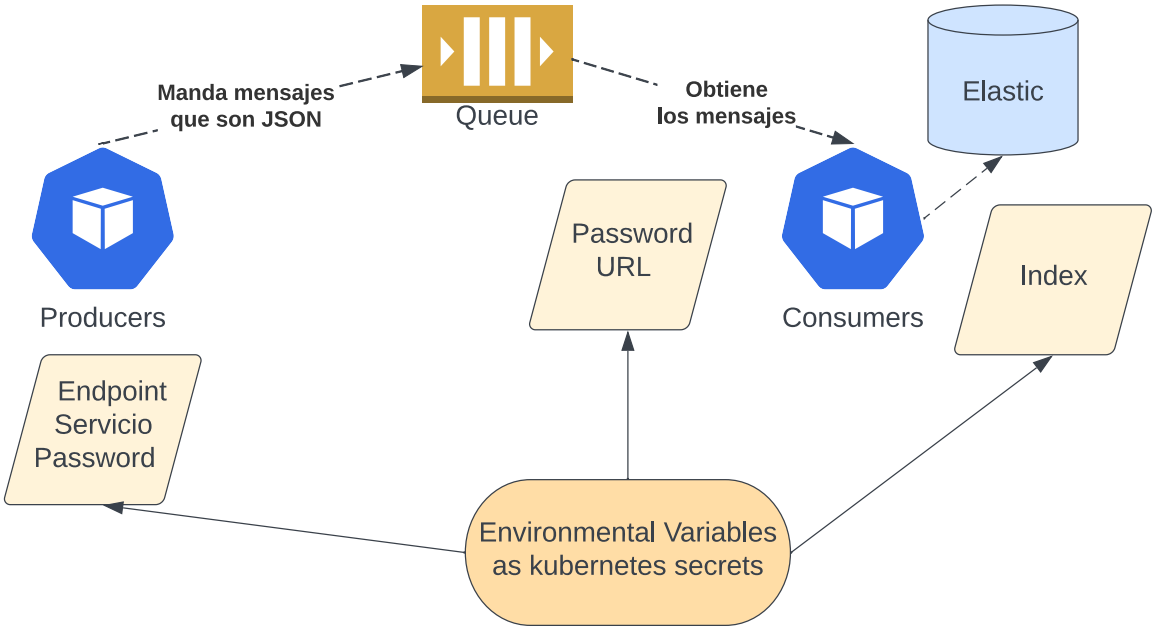
```
docker build -t <repositorio>/<nombre-de-la-imagen>
Ejemplo:
docker build -t deansf/proyecto1-bases2/orchestrator
```

Uso de las docker image para el proyecto

¿Como incluirla en un Helm Chart?

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <nombre>
  labels:
    app: <label>
spec:
  replicas: <numero-de-replicas>
  selector:
    matchLabels:
      app: <label>
template:
  metadata:
    labels:
      app: <label>
  spec:
    containers:
    - name: <nombre-imagen>
      image: <nombre-imagen>
      env:
        - name: <nombre-variable>
          value: <valor-variable>
        - name: <nombre-variable>
          valueFrom:
            secretKeyRef:
              name: <nombre-secret>
              key: <nombre-key>
              optional: <condicional>
```

Explicación de la cola



Codigo de ejemplo en rabbitMQ

Ejemplos salidos de RabbitMQ

Send

```
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

Receiver

```
#!/usr/bin/env python
import pika, sys, os

def main():
    connection =
pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    channel.queue_declare(queue='hello')

    def callback(ch, method, properties, body):
        print(" [x] Received %r" % body)

    channel.basic_consume(queue='hello', on_message_callback=callback,
auto_ack=True)

    print(' [*] Waiting for messages. To exit press CTRL+C')
    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```