Despliegue de una aplicación FiL

Contenido

[Introducción 2](#_Toc69118522)

[Estructura de aplicación FiL 2](#_Toc69118523)

[Imagen base 2](#_Toc69118524)

[Google Container Registry 2](#_Toc69118525)

[Deploy 2](#_Toc69118526)

[Pod 3](#_Toc69118527)

[Service 3](#_Toc69118528)

[Docker Compose y Kompose 4](#_Toc69118529)

[Creación de la aplicación FiL 4](#_Toc69118530)

[Dockerfile 4](#_Toc69118531)

[Docker Compose 6](#_Toc69118532)

[Despliegue de la aplicación FiL 7](#_Toc69118533)

[Análisis previo 7](#_Toc69118534)

[Creación de archivos de despliegue 8](#_Toc69118535)

[Despliegue en Kubernetes 8](#_Toc69118536)

[Monitorización de la aplicación FiL 9](#_Toc69118537)

[Ejemplo práctico 9](#_Toc69118538)

[Creación de la aplicación práctica 9](#_Toc69118539)

[Despliegue de la aplicación practica 12](#_Toc69118540)

# Introducción

Para desplegar una aplicación fog-in-the-loop existen varias etapas. Nosotros vamos a desplegar aplicaciones encapsuladas en contenedores Docker en un clúster Kubernetes. Obviamente, primero tendremos que tener el clúster bien configurado.

Antes de ponernos a crear y desplegar aplicaciones, analizaremos las tecnologías que usamos y la estructura de este tipo de aplicaciones.

# Estructura de aplicación FiL

En una aplicación FiL aparecen varios conceptos:

## Imagen base

Esta es la base sobre la que se construirá la aplicación. Debe tener todo lo necesario para que las aplicaciones puedan ejecutarse.

De un archivo llamado *Dockerfile* vamos a poder crear imágenes base para aplicaciones. En este proceso de crear la imagen, es donde se guardará el código fuente y se instalará lo necesario (Python, Ubuntu) para la ejecución de la aplicación.

## Google Container Registry

Este es el repositorio online donde se almacenarán las imágenes base. Cuando creemos las imágenes en nuestro dispositivo, las subiremos a este repositorio y ya no tendremos que tenerlas en el dispositivo.

Cuando se vaya a desplegar una aplicación se le asignará un nodo del clúster. Este nodo descargará la imagen base desde este repositorio online y construirá la aplicación desde utilizando esa imagen.

## Deploy

Si queremos desplegar una aplicación que esté arrancada en todo momento, utilizaremos los Deployments. Al crear un Deployment le decimos a Kubernetes que queremos una nueva aplicación que esté siempre en el sistema. Si por lo que sea la aplicación cae, ya sea por un fallo de la misma o porque el nodo se cae, por ejemplo.

Kubernetes se encargará de crear una nueva instancia de esa aplicación, para que siempre haya mínimo una. Se pueden poner las réplicas que queramos para cada aplicación.

## Pod

Es la unidad mínima de Kubernetes y la que constituye la instancia de la aplicación en Kubernetes. Un pod representa una instancia única de un proceso en ejecución en tu clúster. Los pods contienen uno o más contenedores, como los contenedores de Docker.

## Service

Cuando queramos exponer una aplicación con las demás, para que todas puedan comunicarse con ella, además de un Deployment crearemos un Service. Un Service habilita que cualquier aplicación, estando en cualquier nodo del cluster, pueda comunicarse con nuestra aplicación.

Se podrán comunicar mediante el nombre del servicio de nuestra aplicación, pongamos *serviceApp1*. Así, cuando cualquier otra aplicación mande un mensaje al elemento *serviceApp1* se enlazará ese nombre con la dirección IP del pod. Gracias a esto, da igual que los pod se caigan y se reconstruyan, podremos comunicarnos siempre con ella.

Si la comunicación va a desarrollarse solo dentro del cluster, el servicio será del tipo **ClusterIP**. En cambio, si queremos exponer alguna aplicación fuera de nuestro cluster, utilizaremos el servicio **NodePort**.

spec:

type: NodePort

ports:

- name: "8080"

port: 8080

targetPort: 8080

nodePort: 30808

Así, la aplicación quedará expuesta a fuera del cluster. Se podrá acceder a ella mediante la dirección IP del nodo y el puerto que se haya seleccionado en el apartado del *nodePort*: http://<direccion IP de cualquier nodo>:<nodePort>

## Docker Compose y Kompose

Con la herramienta Docker Compose, podremos definir aplicaciones de varios contenedores, es decir, aplicaciones de componentes múltiples. Con un simple comando podremos crear varias imágenes base a la vez, o arrancar una aplicación FiL con varios componentes.

La herramienta Kompose utilizando la definición de la aplicación FiL en Docker Compose, traducirá toda esa definición a archivos de despliegues de Kubernetes. Asi, si necesitamos exponer un componente creará su archivo YAML para el *Deployment* y el *Service*.

# Creación de la aplicación FiL

En esta fase, crearemos todas las imágenes base de los componentes de la aplicación FiL y los almacenaremos para que queden listos para su despliegue. Da igual donde tengamos los archivos necesarios para crear la imagen base, pero deberá haber una congruencia entre el sitio donde estén los archivos y el path de Dockerfile, Docker Compose, etc.

El primer paso será analizar cada componente de la aplicación. Deberemos tener en cuenta varios factores, como el lenguaje de programación en el que se desarrollarán o si necesitan algún fichero de configuración, etc. Cuando tengamos todos los puntos claros, podremos comenzar a crear los contenedores para nuestros componentes.

## Dockerfile

Por cada componente necesitaremos un fichero esencial: **Dockerfile**. Aquí se definirá la construcción de cada componente: se copiarán los archivos con el código fuente dentro de la imagen, se descargará e instalará la imagen base… También se añadirá el comando que se ejecutará cuando se arranque el contenedor.

Todo contenedor Docker necesita una imagen base en la cual va a instalar los necesario para la aplicación. Por ejemplo, si es un servidor Flask desarrollado en Python, desde la imagen base de Python, instalará el paquete flask. Por esto mismo comentábamos anteriormente que hay que tener bien analizados los componentes. En este caso, como este componente está desarrollado en Python, se utilizará la imagen base *python:3* desde la cual se construirá el contenedor.

Ejemplo de Dockerfile:

FROM python:3

RUN pip install flask

COPY pqp.py .

ENV FLASK\_APP=pqp.py

ENV FLASK\_RUN\_PORT=6000

CMD flask run --host=0.0.0.0

Si se quiere crear imágenes de contenedores individualmente, podremos hacerlo mediante estos comandos. Las dos opciones son válidas.

> docker build –t <tag de la imagen> -f <path hacia el Dockerfile> .

> docker build –t <tag de la imagen> -f <nombre archivo Dockerfile> <path del directorio del Dockerfile>

Si en la carpeta donde ejecutamos el comando tenemos un único elemento con nombre Dockerfile, podremos simplificar el comando:

> docker build –t <tag de la imagen> .

Si el tag de la imagen lo creamos con la estructura de Google Container Registry, podremos subirlo al repositorio online:

> docker push <tag de la imagen GCR>

En el caso de que hayamos elegido otro nombre, se podrá modificar para poder subirlo al repositorio.

> docker tag <tag del usuario> <tag de la imagen GCR>

En el caso de Google Container Registry el path o tag de la imagen deberá tener una estructura concreta:

gcr.io/<id del proyecto>/<nombre del repositorio>:<nombre de la etiqueta>

## Docker Compose

Por cada aplicación FiL necesitaremos un archivo llamado ***docker-compose.yml***. Este es en el que se definirá cada componente de la aplicación. Por cada componente podremos añadir las opciones que nos brinde Docker Compose. En nuestros casos los más utilizados son: *image* (para saber el nombre de la imagen), *build* (para la construcción el contenedor), *ports* (para asignar los puertos por los que se comunicarán las demás aplicaciones con esta) …

En esta sección será en la que especificaremos todos los componentes de nuestra aplicación y sus definiciones.

version: "2"

services:

source:

build:

context: .

dockerfile: Dockerfile\_source

image: gcr.io/clusterekaitz/appfil1:source

container\_name: source

environment:

- CLIENT\_NAME=source

- TOPIC=#

- SINK\_NAME=sink

- TRANSFORM\_FILE=mqtt2http.py

sink:

build:

context: ./src

dockerfile: Dockerfile\_sink

image: gcr.io/clusterekaitz/appfil1:sink\_tomcat

container\_name: sink

ports:

- 8080:8080

environment:

- SERVICE\_TYPE=updatePLCInfo

Como se puede ver, se añaden variables de entorno. Esta es la forma con la que podremos parametrizar las aplicaciones en contenedores. El código fuente está dentro de la imagen, por lo que, para poder, por ejemplo, cambiar valores de variables, usaremos las variables de entorno del pod. Desde el código conseguiremos estas variables, y tendrán los valores que le hayamos añadido en este apartado de Docker Compose.

Aquí es donde acabaría la primera fase de la creación de la aplicación. Teniendo todos los archivos en su sitio (siguiendo el caso anterior, el archivo *Dockerfile\_sink* debe estar en la carpeta *src*, la cual está en el directorio donde ejecutamos el comando) podremos crear las imágenes base de los componentes.

Para crear todas las imágenes de los componentes, vale con ejecutar este comando en el mismo directorio donde se encuentra el archivo *docker-compose.yml*:

> docker-compose build [1]

Una vez tengamos todas las imágenes en el dispositivo donde hemos ejecutado el comando (si se desean ver basta con ejecutar: > docker images), podremos subirlas al repositorio online de Google Container Registry. Se podrán subir las imágenes por repositorios, es decir, si nosotros tenemos 4 imágenes dentro de un repositorio, con este comando se subirán todas a la vez. En el caso de tener más de un repositorio, habrá que repetir el comando por cada uno.

> docker push gcr.io/<id proyecto>/<nombre repositorio> [2]

# Despliegue de la aplicación FiL

## Análisis previo

A la hora de desplegar la aplicación en el cluster Kubernetes, ya habremos creado las imágenes de cada componente y se habrán almacenado en el repositorio online. El procedimiento será el siguiente, cuando nosotros realizamos una petición de despliegue, el nodo maestro asignará un nodo a cada componente. El elemento *kubelet* de esos nodos descargará las imágenes de los repositorios online y creará la instancia de la aplicación a partir de ellos.

Un dato a tener en cuenta es que, si el nodo donde se va a desplegar la aplicación ya tiene dicha imagen descargada, no lo volverá a hacer, aunque la imagen del repositorio online sea una versión más actualizada. Por ello, si queremos eliminar una imagen de un nodo, tendremos que ir a ese nodo y ejecutar este comando:

> docker rmi <tag de la imagen>

## Creación de archivos de despliegue

Después del análisis previo, podremos crear los archivos de despliegue para Kubernetes. Estos archivos deben ser de tipo YAML. Para conseguirlos, utilizaremos la herramienta **Kompose**. Esta herramienta nos permitirá conseguir los archivos YAML desde el *docker-compose.yml*. La función de Kompose es analizar las definiciones de los componentes en el archivo *docker-compose.yml* y crear los YAML necesarios. Por ejemplo, si una aplicación necesita ser visible para las demás, Kompose creará un archivo para la creación de un servicio (*service*), a parte del de despliegue (*deployment*).

Este proceso es muy sencillo, basta con ejecutar este comando en el directorio del archivo *docker-compose.yml*.

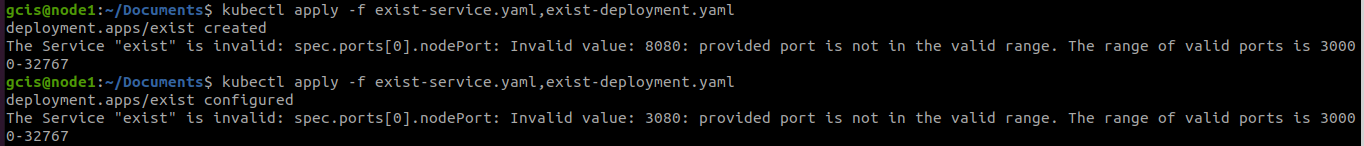
> kompose convert [3]

Esto creará todos los archivos YAML en el mismo directorio. Después, podremos mover estos archivos a donde queramos. Siempre creará los archivos con el nombre del servicio especificado en Docker Compose más *“-deployment.yaml”* o *“-service.yaml”* dependiendo de lo que necesite.

## Despliegue en Kubernetes

Si necesitamos exponer algún componente fuera del cluster deberemos cambiar el archivo de creación de servicio YAML manualmente. Así, buscaremos el archivo “<nombre componente>-service.yaml” y añadiremos dos nuevos datos. El primero, cuando comienza el apartado de *spec* añadiremos *type: NodePort* y el segundo, debajo de cada *targetPort* añadiremos el puerto a exponer fuera del cluster *nodePort: <puerto a exponer>*.

El puerto a exponer debe tener un valor entre 3000 y 32767, si no, nos dará error. Es el único rango que nos ofrece Kubernetes.



Una vez tengamos los archivos YAML finales, podremos desplegarlos en Kubernetes. Podremos hacerlo uno a uno, o desplegar todos los archivos YAML que se encuentren dentro de un directorio.

> kubectl apply –f app1-deployment.yaml,app2-deployment.yaml,app1-service.yaml

> kubectl apply –f /path al directorio/directorio [4]

# Monitorización de la aplicación FiL

Una vez tengamos desplegada la aplicación en nuestro cluster, hay varias formas de monitorizarla. Como hemos comentado anteriormente los pod se destruyen y construyen constantemente. Kubernetes cada vez que se vuelve a crear un pod le asigna un nombre distinto. Para visualizar todos los pods actuales del cluster basta con este comando:

> kubectl get pods -o wide

Si queremos ver la consola en directo de un componente tendremos que fijarnos en su nombre con el comando anterior y ejecutar este otro:

> kubectl logs -f <nombre del pod>

Si queremos gestionar un despliegue de un componente en concreto podremos hacerlo mediante este comando:

> kubectl rollout delete/restart deploy <nombre del deploy>

También podremos entrar dentro del pod:

> kubectl exec –it <nombre del pod> -- sh

# Ejemplo práctico

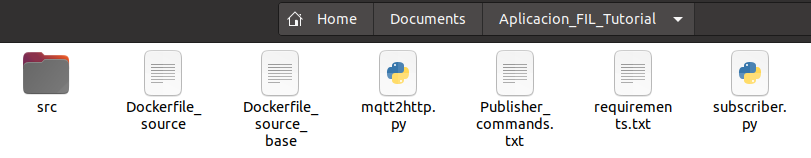
## Creación de la aplicación práctica

Vamos a seguir un ejemplo de una aplicación FiL que consta de dos componentes: source y sink. Primero, deberemos analizar cada componente. En este caso los dos van a estar desarrollados en Python, pero necesitarán elementos diferentes.

El componente source consta de un suscriptor MQTT y un transformador que envía los datos vía HTTP. Para que pueda ejecutarse necesitará instalar el paquete *paho-mqtt* para el suscriptor y *requests* para gestionar, en este caso enviar, mensajes HTTP. En el caso de sink utilizará la imagen base de Tomcat para poder arrancar el servidor.

Teniendo analizado lo que necesitan los componentes podremos desarrollar el código fuente, si es necesario un código fuente, y se podrá desarrollar cuando se quiera (se puede desarrollar antes de análisis).

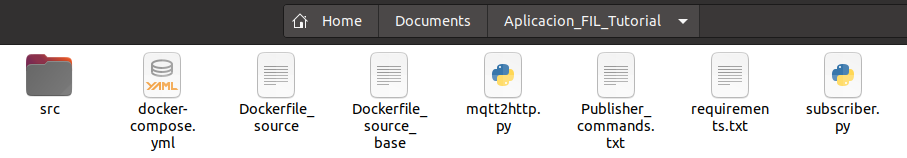
Al finalizar esta etapa, en una carpeta en nuestro dispositivo tendremos todos los archivos necesarios.



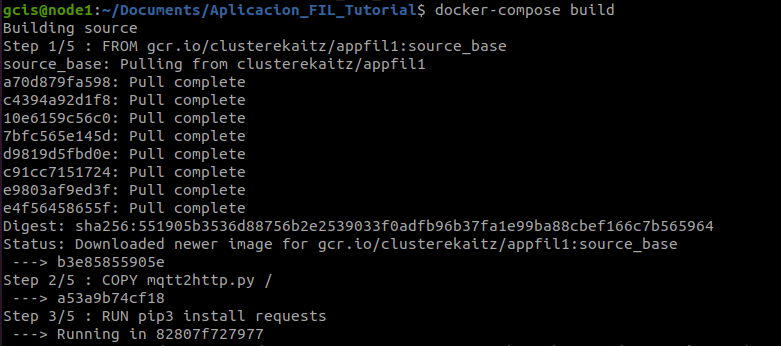
Ahora nos pondremos con el contenido del ***docker-compose.yml***, el cual es el mismo analizado en el anterior apartado. Aquí es donde daremos forma a nuestra aplicación FiL. Añadiremos y definiremos los componentes deseados. Las dos primeras líneas siempre serán las mismas: la versión de Docker Compose y “*services:”*. A partir de aquí, cada vez que añadamos un componente le pondremos primero el *nombre*, *context* para crear la imagen con el Dockerfile, *image* para especificar el tag de la imagen.

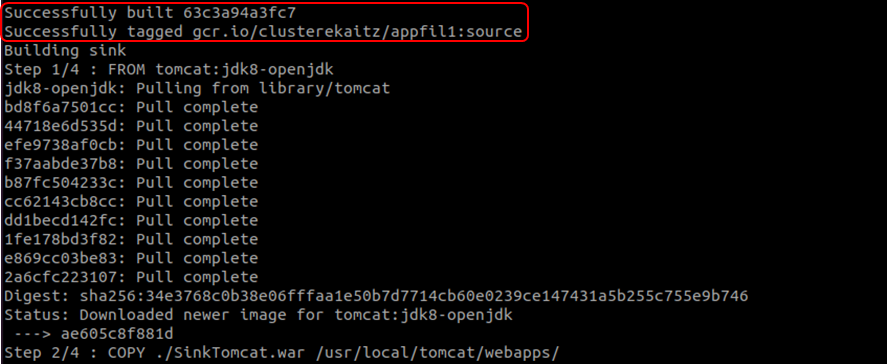
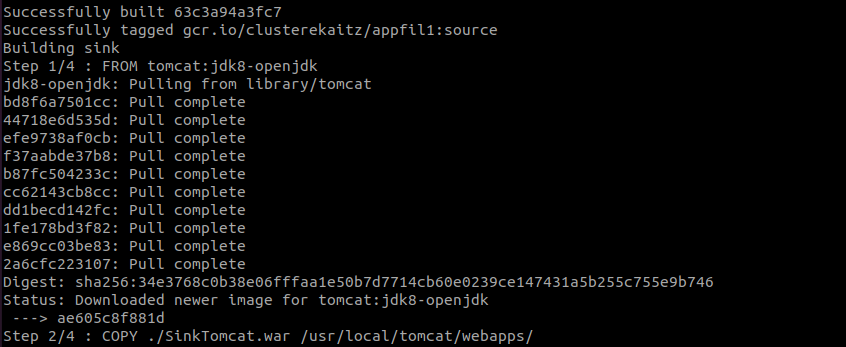
Hay otras opciones opcionales para parametrizar los componentes: *container\_name* para añadirle un nombre al contenedor, *environments* para añadir variables de entorno, *ports* para asignar puertos de entrada y salida…

Cuando tengamos desarrollado el archivo tendremos todo lo necesario en el directorio:

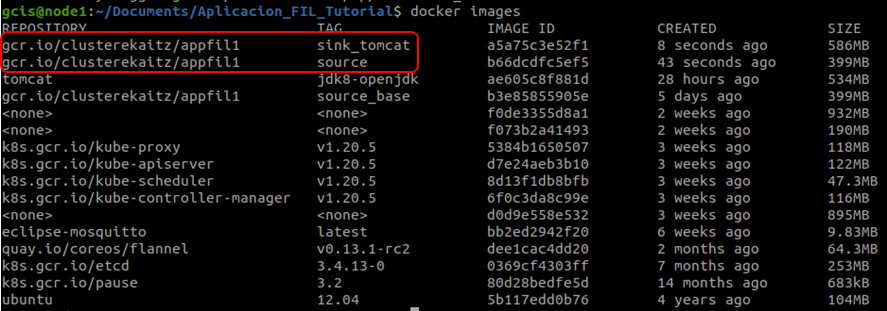
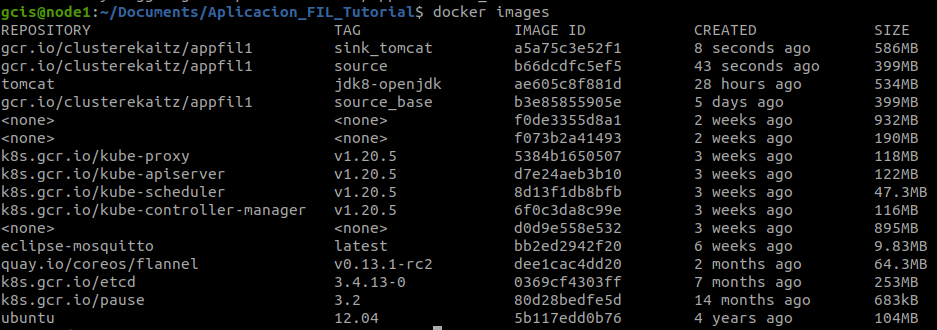


En este punto, ya podremos crear las imágenes base de los componentes. Para ello, como ya hemos comentado utilizaremos la herramienta Docker Compose para poder crear todas las imágenes a la vez (comando [1]).

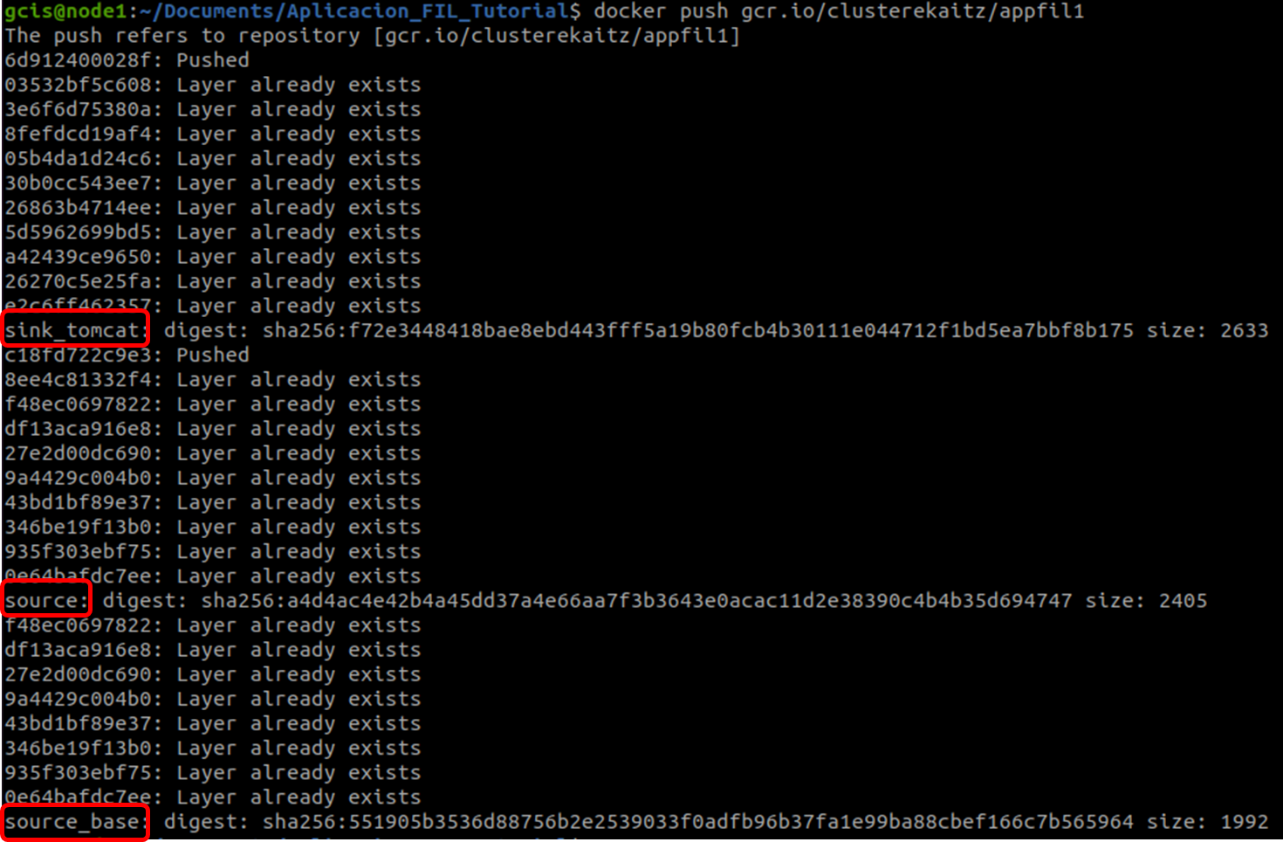




Como se puede apreciar, crea las imágenes definidas en el archivo *docker-compose.yml* y les da el nombre que hayamos definido. En nuestro caso los nombres son *gcr.io/clusterekaitz/appfil1:source* y *gcr.io/clusterekaitz/appfil1:sink* para poder subirlos directamente el registro de imágenes online. (comando [2]) Podremos comprobar si se han creado las imágenes en nuestro dispositivo:

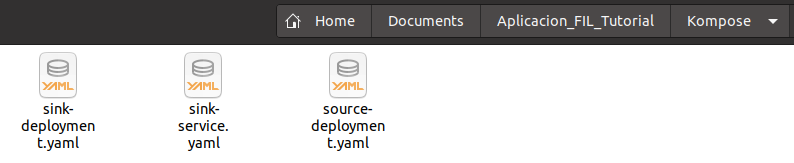


Ahora subiremos las imágenes que queramos al registro online, en nuestro caso. Google Container Registry. Podremos subirlas una a una o por repositorio. Los componentes que hemos creado los hemos definido en el mismo repositorio, así que lo subiremos al registro:

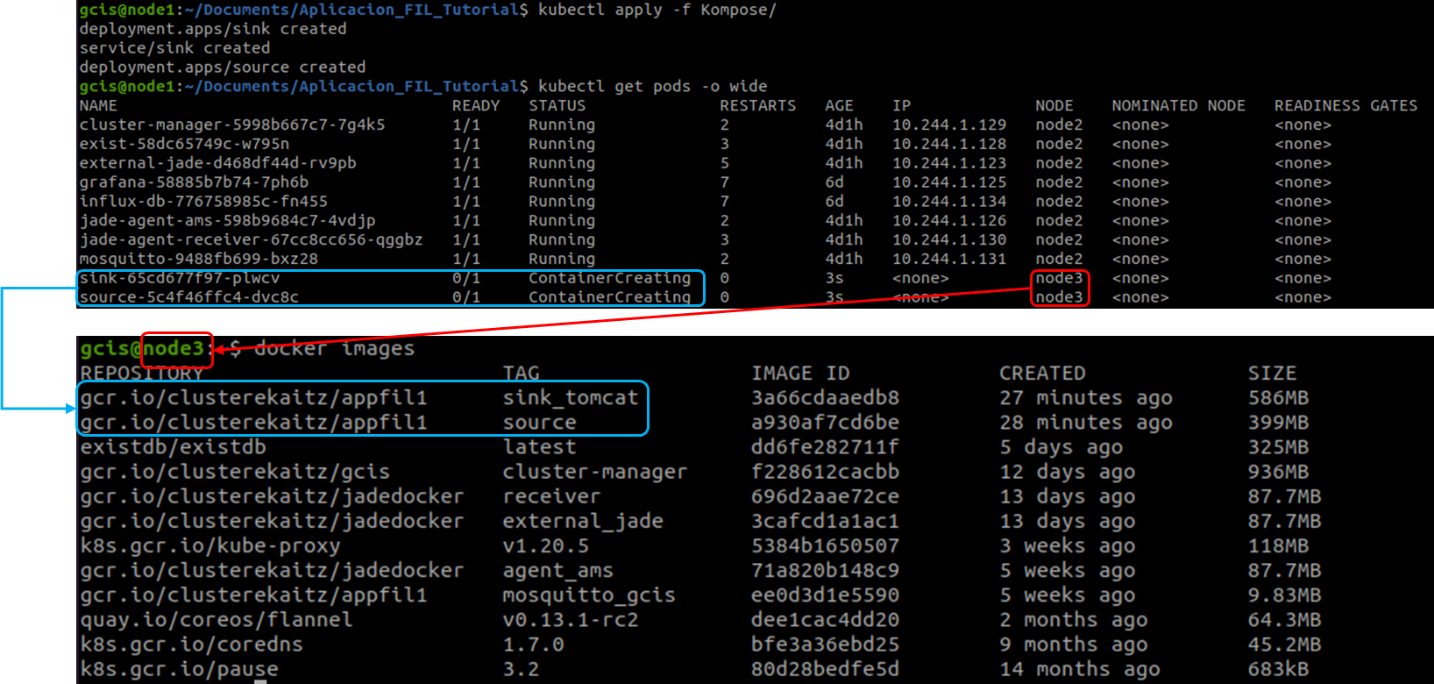


## Despliegue de la aplicación practica

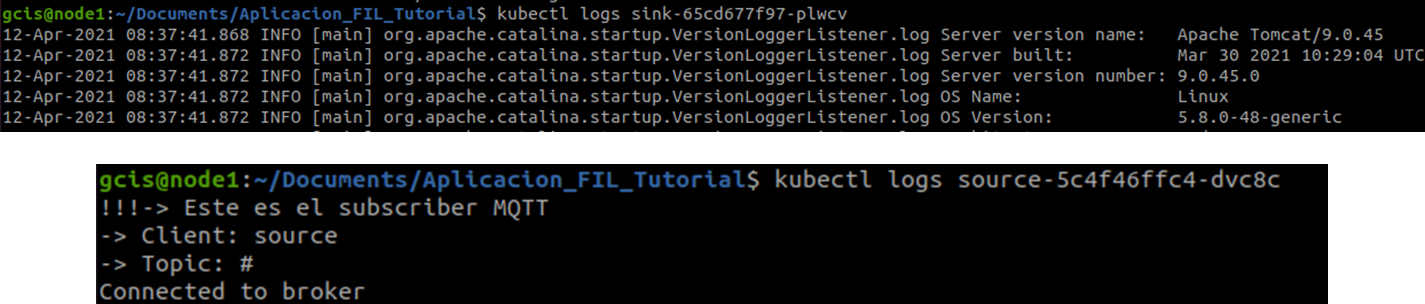
Ya que tenemos todas las imágenes base de los componentes en el registro, podremos desplegar la aplicación. Para ello utilizaremos la herramienta Kompose para crear los archivos de despliegue (comando [3]):



Con todo preparado, desplegaremos la aplicación en el cluster Kubernetes (comando [4]):



Si analizamos las consolas de los componentes, podremos ver que se han arrancado correctamente.



Por último, si queremos eliminar todos los despliegues, utilizaremos los mismos archivos YAML de despliegue:

