Introducción a Docker  
UD 08. Caso práctico 01 - Aplicación Flask con Kubernetes

short line

Autor: Sergi García Barea

Actualizado Abril 2022

Licencia

**Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA)**: No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

**1.** [**Introducción**](#_dbh0n1vac4c8) **3**

**2.** [**Paso 0: iniciar MiniKube**](#_3b05p2f899fh) **3**

**3.** [**Paso 1: Aplicación “app.py” y “Dockerfile” del caso práctico**](#_vyhbfp4t666x) **3**

**4.** [**Paso 2 (Comandos): Desplegando una aplicación mediante comandos**](#_m1jtbkgv3d98) **4**

**5.** [**Paso 4: escalando nuestro despliegue**](#_9p0xnot11ut4) **6**

**6.** [**Paso 5 (Fichero YAML):Desplegando la aplicación mediante ficheros YAML**](#_v5ot8tcfc7uh) **7**

**7.** [**Paso 6: eliminando lo creado**](#_okij6l7ljwiv) **9**

**8.** [**Bibliografía**](#_odopdfh4u6on) **9**

UD08. Caso práctico 01

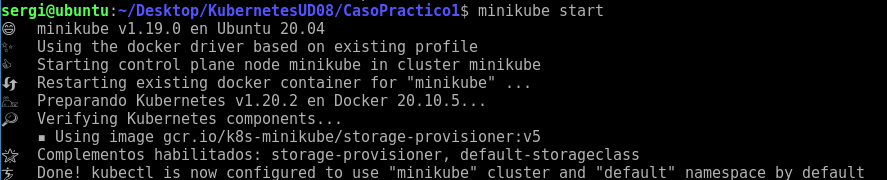
# Introducción

En este caso práctico vamos a poner en marcha una aplicación servidor con “***Python*”** usando “***Flask*”** <https://flask.palletsprojects.com/en/1.1.x/>. En el caso práctico desplegamos esta aplicación usando “***Kubernetes***” y “***MiniKube***”.

# Paso 0: iniciar MiniKube

Antes de empezar el caso práctico, debemos poner en marcha nuestro clúster con:

| minikube start |
| --- |



Una vez puesto en marcha, podemos proseguir con el caso práctico.

# Paso 1: Aplicación “app.py” y “Dockerfile” del caso práctico

Para el caso práctico, utilizaremos una sencilla aplicación web con “***Python***” y “***Flask***” que simplemente imprimirá el “***hostname***” (formado por identificador del contenedor) de quien está sirviendo la aplicación. El contenido de “***app.py***” es el siguiente:

| **from flask import Flask, escape, request import socket app = Flask(\_\_name\_\_)  @app.route('/') def get\_hostname():  return "Aplicación servida desde hostname: "+socket.gethostname()** |
| --- |

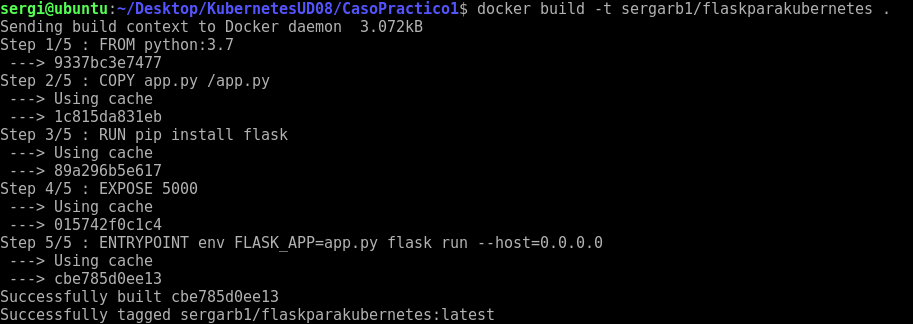
El contenido del fichero “***Dockerfile***” que incluimos comentado, es el siguiente:

| *#Utilizamos la imagen con Python 3.7* **FROM** python:3.7 *#Copiamos del anfitrión a la imagen la aplicación* **COPY** app.py /app.py *#Instalamos la biblioteca Flash* **RUN** pip install flask *#Exponemos el puerto 5000* **EXPOSE** 5000 *#Indicamos que se ejecute la aplicación al iniciar el contenedor* **ENTRYPOINT** env FLASK\_APP=app.py flask run --host=0.0.0.0 |
| --- |

Con esos dos ficheros, crearemos la imágen en nuestra máquina usando un comando similar a:

| docker build -t sergarb1/flaskparakubernetes . |
| --- |

Tras lanzar esta orden, obtendremos algo similar a:



Una vez creada la imagen, podríamos crear un simple contenedor utilizando un comando similar a

| docker run -d --name miapp -p 5000:5000 sergarb1/flaskparakubernetes |
| --- |

pero en este caso, omitiremos este paso ya que realizaremos el despliegue usando “***Kubernetes***”.

Una vez creada, deberemos subirla a un registro como “***Docker Hub***”. Podéis subirla a vuestra cuenta o por comodidad simplemente usar una imagen subida a mi cuenta de “***Docker Hub***”.

**Enlace a la imagen:** [**https://hub.docker.com/repository/docker/sergarb1/flaskparakubernetes**](https://hub.docker.com/repository/docker/sergarb1/flaskparakubernetes)

📖 **Importante:** se puede configurar para que la busque localmente, pero va “en contra” del propósito de “***Kubernetes***” (la idea es que podamos desplegar en un cluster distribuido, donde cada nodo pueda estar en una máquina).

# Paso 2 (Comandos): Desplegando una aplicación mediante comandos

Para desplegar esta aplicación en “***Kubernetes***” mediante comandos, en primer lugar crearemos un “***Pod***” en nuestro clúster mediante:

| kubectl create deployment midespliegue --image=sergarb1/flaskparakubernetes --port=5000 |
| --- |

Obteniendo algo similar a:



Con esta orden estamos:

* Creando un “***Pod***” con nombre “***miapp***”.
* Indicando que el “***Pod***” utiliza la imagen “***sergarb1/flaskparakubernetes***”.
* Se expone el puerto 5000 de la aplicación.

Tras ello, podremos comprobar el estado de los “***Pod***” creado usando:

| kubectl get pods |
| --- |

Obteniendo algo similar a:



Una vez creado nuestro “***Pod***”, vamos a exponerlo como servicio para que se pueda acceder a nuestra aplicación. Podemos hacerlo con el siguiente comando:

| kubectl expose deployment midespliegue --type=LoadBalancer --name=midespliegue-http |
| --- |

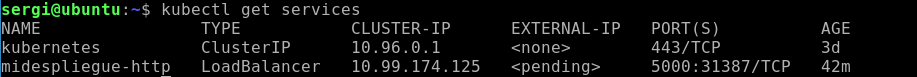
Obteniendo como respuesta:



Una vez hecho esto si examinamos los servicios:

| kubectl get services |
| --- |

Obteniendo algo similar a:

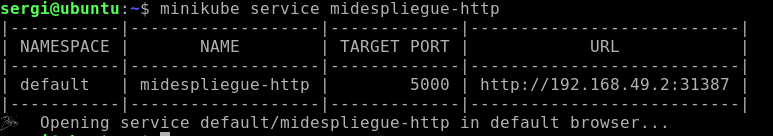


Si observamos detenidamente, la IP Externa de nuestro servicio está “***Pending***” y no lo tenemos expuesto directamente. Esto es debido a que todo el clúster “***Kubernetes***” está dentro de la máquina virtual de “***MiniKube***”. Para acceder al contenido, tenemos dos formas:

**Forma 1:** accederemos a la IP de “***MiniKube***” y nos expondrá el servicio en un puerto aleatorio.

| minikube service midespliegue-http |
| --- |

Tras lanzar este comando, se nos abrirá un navegador accediendo al servicio en uno de los puertos que expone “***MiniKube***” y aparecerá un texto similar a:



Tras ello podemos observar que nuestra aplicación está siendo servida:

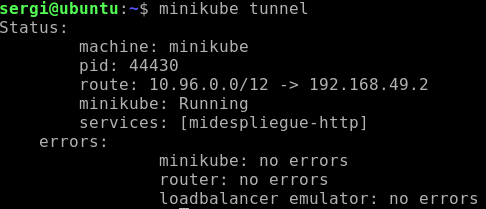


**Forma 2:** expondremos el servicio con la IP de “***MiniKube***” y accederemos a él.

Para hacer esto, en una terminal aparte lanzaremos el siguiente comando

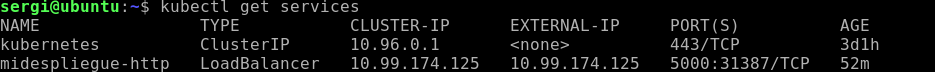
| minikube tunnel |
| --- |

Este comando se ejecutará en la terminal “de forma indefinida” y **mientras esté en funcionamiento**, establecerá un túnel para acceder al servicio. Veremos algo similar a:



Si mientras este comando está en ejecución hacemos

| kubectl get services |
| --- |



Ya observaremos una IP. En este ejemplo, accediendo al puerto 5000 de la IP del balanceador (en este ejemplo http://10.99.174.125:5000) accederemos a la aplicación Flask desplegada.

# Paso 4: escalando nuestro despliegue

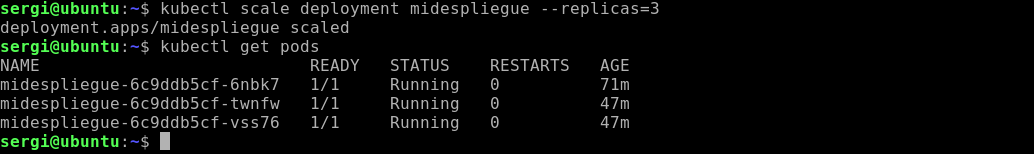
Si queremos escalar el número de “***Pods***” de nuestro despliegue, podemos hacerlo de forma dinámica mediante comandos. Por ejemplo:

| kubectl scale deployment midespliegue --replicas=3 |
| --- |

Establecerá 3 réplicas. Si tras lanzarlo vemos los “***Pods***”:

| kubectl get pods |
| --- |

Observamos algo similar a:

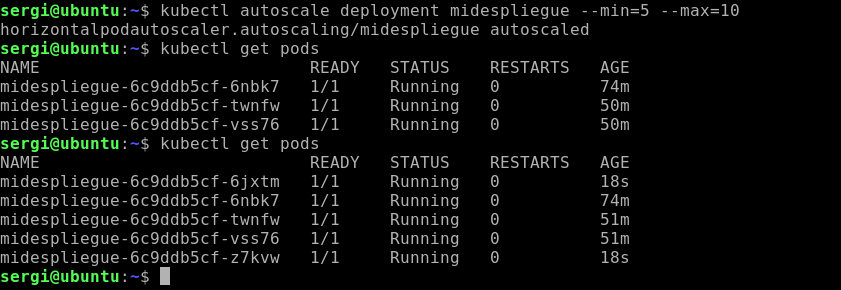


Donde se han creado 3 réplicas. Si accedemos a la máquina como se describió en el paso anterior, veremos que cada vez nos atenderá uno de los “***Pods***” gracias al balanceo de carga.

Si queremos que “***Kubernetes***” realice un autoescalado, nada tan fácil como ejecutar:

| kubectl autoscale deployment midespliegue --min=5 --max=10 |
| --- |

Observamos la siguiente imagen:



Se observa que previo al autoescalado, había 3 “***Pods***”. A los pocos segundos, se ha aplicado el auto-escalado (con un mínimo de 5, pero la posibilidad de auto-escalarse a 10).

Si comprobamos el acceso como se explica en el “***Punto 3***”, veremos que nos sirve la aplicación hasta 5 contenedores distintos.

# Paso 5 (Fichero YAML):Desplegando la aplicación mediante ficheros YAML

Antes de empezar, si hemos realizado el paso anterior, deberemos eliminar tanto el despliegue como el servicio donde hemos expuesto el mismo con los comandos:

| kubectl delete deployment midespliegue |
| --- |

| kubectl delete service midespliegue-http |
| --- |

También deberemos eliminar el autoescalado aplicado al despliegue con:

| kubectl delete horizontalpodautoscaler midespliegue |
| --- |

Vamos a definir la configuración presentada al final del “**Paso 4**” utilizando un fichero **YAML** comentado “***deployment.yaml***” con el siguiente contenido.

| *#Indicamos la versión de la API apiVersion: apps/v1 #Indicamos que este fichero es de un despliegue kind: Deployment #Metadatos del despliegue metadata:  name: midespliegue #Características del despliegue spec:  # Al inicio 3 réplicas (luego si queremos activamos autoescalado)  replicas: 3  #Selector de los pods  selector:  matchLabels:  app: midespliegue  #Plantilla de los pods  template:  #Metadatos de los pods  metadata:  labels:  app: midespliegue  #Caracteristicas de los pods  spec:  #Contenedor del pod  containers:  #Nombre, imagen y puerto expuesto  - name: miapp  image: sergarb1/flaskparakubernetes  ports:  - containerPort: 5000* |
| --- |

Una vez listo, podemos lanzar nuestro despliegue usando el comando:

| kubectl apply -f "deployment.yaml" |
| --- |

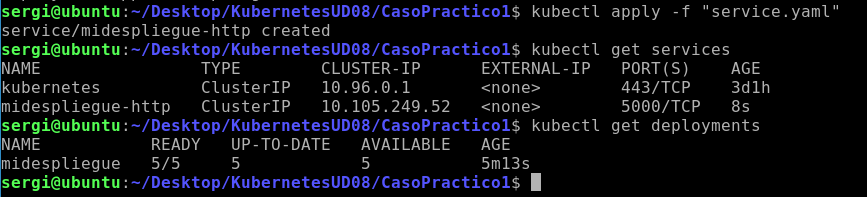
Con esto habremos creado nuestro despliegue. Observaremos algo similar a:



Tras ello, deberemos crear nuestro servicio con un fichero “***service.yaml***”

| *#Versión de la API* apiVersion: v1 *#Definimos un servicio* kind: Service *#Metadatos del servicio* metadata:  name: midespliegue-http *#Características del servicio* spec:  *#Tipo de servicio*  type: LoadBalancer  *#Puerto a exponer*  ports:  - port: 5000  targetPort: 5000  *#A que aplica el servicio, busca los que coincidan con app: midespliegue para servir*  selector:  app: midespliegue |
| --- |

Si todo ha ido bien, tendremos algo similar a:



Si comprobamos el acceso como se explica en el “***Punto 3***”, veremos que nos sirve la aplicación correctamente.

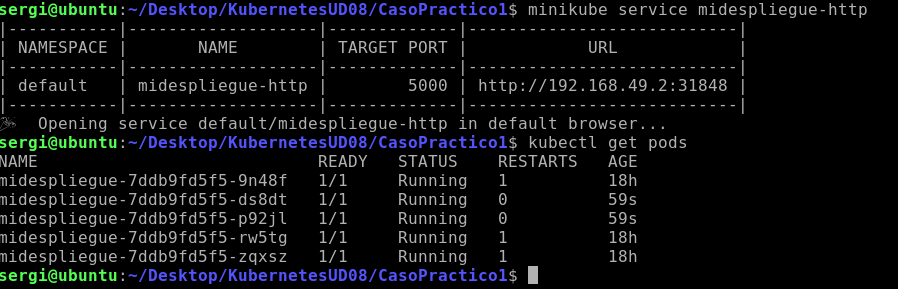
Por último, definiremos mediante un fichero “***autoscale.yaml***” el auto-escalado del despliegue. El contenido del fichero será el siguiente:

| apiVersion: autoscaling/v1 *#Tipo autoescalado horizontal* kind: HorizontalPodAutoscaler metadata:  name: autoescalado spec:  *#Indicamos a quien se aplica el auto-escalado*  scaleTargetRef:  apiVersion: apps/v1  kind: Deployment  name: midespliegue  *#Minimo y maximo de réplicas*  minReplicas: 5  maxReplicas: 10  *#Máximo de CPU a usar durante el auto-escalado*  targetCPUUtilizationPercentage: 50 |
| --- |

Una vez listo, podemos lanzar nuestro despliegue usando el comando:

| kubectl apply -f "autoscale.yaml" |
| --- |

Para comprobar que el autoescalado se ha realizado correctamente, deberemos realizar una petición al servicio del despliegue, de forma similar a lo comentado en el “***Paso 3***”. Tras esto, escalara y veremos que el sistema ha autoescalado a 5 “***Pods***”.



# Paso 6: eliminando lo creado

Si queremos eliminar todos los elementos creados, podemos hacerlo con los siguientes comandos:

| kubectl delete deployment midespliegue kubectl delete service midespliegue-http kubectl delete HorizontalPodAutoscaler autoescalado |
| --- |

# Bibliografía

[1] Kubernetes <https://kubernetes.io/>

[2] Kubernetes docs <https://kubernetes.io/docs/home/>

[3] MiniKube <https://minikube.sigs.k8s.io/docs/>