

Introducción a Docker

UD 08. Caso práctico 01

- Aplicación Flask con Kubernetes



Fons Social Europeu

L'FSE inverteix en el teu futur

Autor: Sergi García Barea

Actualizado Abril 2021

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	3
2. Paso 0: iniciar MiniKube	3
3. Paso 1: Aplicación "app.py" y "Dockerfile" del caso práctico	3
4. Paso 2 (Comandos): Desplegando una aplicación mediante comandos	4
5. Paso 4: escalando nuestro despliegue	6
6. Paso 5 (Fichero YAML):Desplegando la aplicación mediante ficheros YAML	7
7. Paso 6: eliminando lo creado	9
8. Bibliografía	9

UD08. CASO PRÁCTICO 01

1. 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”**.

2. PASO 0: INICIAR MINIKUBE

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

```
minikube start
```

```
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ minikube start
minikube v1.19.0 en Ubuntu 20.04
Using the docker driver based on existing profile
Starting control plane node minikube in cluster minikube
Restarting existing docker container for "minikube" ...
Preparando Kubernetes v1.20.2 en Docker 20.10.5...
Verifying Kubernetes components...
  Using image gcr.io/k8s-minikube/storage-provisioner:v5
Complementos habilitados: storage-provisioner, default-storageclass
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

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

3. 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 Flask
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 imagen en nuestra máquina usando un comando similar a:

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

Tras lanzar esta orden, obtendremos algo similar a:

```
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ docker build -t sergarb1/flaskparakubernetes .
Sending build context to Docker daemon 3.072kB
Step 1/5 : FROM python:3.7
--> 9337bc3e7477
Step 2/5 : COPY app.py /app.py
--> Using cache
--> 1c815da831eb
Step 3/5 : RUN pip install flask
--> Using cache
--> 89a296b5e617
Step 4/5 : EXPOSE 5000
--> Using cache
--> 015742f0c1c4
Step 5/5 : ENTRYPOINT env FLASK_APP=app.py flask run --host=0.0.0.0
--> Using cache
--> cbe785d0ee13
Successfully built cbe785d0ee13
Successfully tagged sergarb1/flaskparakubernetes:latest
```


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>

 **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).

4. PASO 2 (COMANDOS): DESPLEGANDO UNA APLICACIÓN MEDIANTE COMANDOS

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

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

Obteniendo algo similar a:

```
sergi@ubuntu:~$ kubectl create deployment midespliegue --image=sergarb1/flaskparakubernetes --port=5000
deployment.apps/midespliegue created
```

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:

```
sergi@ubuntu:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
midespliegue-6c9ddb5cf-7zrrc       1/1     Running   0           18s
```

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:

```
sergi@ubuntu:~$ kubectl expose pod miapp --type=LoadBalancer --name=miapp-http
service/miapp-http exposed
```

Una vez hecho esto si examinamos los servicios:

```
kubectl get services
```

Obteniendo algo similar a:

```
sergi@ubuntu:~$ kubectl get services
NAME                TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
kubernetes          ClusterIP     10.96.0.1    <none>        443/TCP          3d
midespliegue-http   LoadBalancer 10.99.174.125 <pending>     5000:31387/TCP   42m
```

Si observamos detenidamente, la IP Externa de nuestro servicio está “**Pending**” y no lo tenemos expuesto directamente. Esto es debido a que todo el cluster “**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 random.

```
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:

```
sergi@ubuntu:~$ minikube service midespliegue-http
|-----|
| NAMESPACE | NAME           | TARGET PORT | URL                               |
|-----|
| default   | midespliegue-http | 5000        | http://192.168.49.2:31387       |
|-----|
🐳 Opening service default/midespliegue-http in default browser...
```

Tras ello podemos observar que nuestra aplicación está siendo servida:



192.168.49.2:31387/

Aplicación servida desde hostname: midespliegue-6c9ddb5cf-twnfw

Forma 2: exponremos 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:

```
sergi@ubuntu:~$ minikube tunnel
Status:
  machine: minikube
  pid: 44430
  route: 10.96.0.0/12 -> 192.168.49.2
  minikube: Running
  services: [midespliegue-http]
  errors:
    minikube: no errors
    router: no errors
    loadbalancer emulator: no errors
```

Si mientras este comando está en ejecución hacemos

```
kubectl get services
```

```
sergi@ubuntu:~$ kubectl get services
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes          ClusterIP   10.96.0.1     <none>         443/TCP          3d1h
midespliegue-http   LoadBalancer 10.99.174.125 10.99.174.125 5000:31387/TCP   52m
```

Ya observaremos una IP. En este ejemplo, accediendo a <http://10.99.174.125:5000> accederemos a la aplicación Flask desplegada.

5. 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:

```
sergi@ubuntu:~$ kubectl scale deployment midespliegue --replicas=3
deployment.apps/midespliegue scaled
sergi@ubuntu:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
midespliegue-6c9ddb5cf-6nbk7        1/1     Running   0           71m
midespliegue-6c9ddb5cf-twnfw        1/1     Running   0           47m
midespliegue-6c9ddb5cf-vss76        1/1     Running   0           47m
sergi@ubuntu:~$
```

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:

```
sergi@ubuntu:~$ kubectl autoscale deployment midespliegue --min=5 --max=10
horizontalpodautoscaler.autoscaling/midespliegue autoscaled
sergi@ubuntu:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
midespliegue-6c9ddb5cf-6nbk7        1/1     Running   0           74m
midespliegue-6c9ddb5cf-twnfw        1/1     Running   0           50m
midespliegue-6c9ddb5cf-vss76        1/1     Running   0           50m
sergi@ubuntu:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
midespliegue-6c9ddb5cf-6jxtm        1/1     Running   0           18s
midespliegue-6c9ddb5cf-6nbk7        1/1     Running   0           74m
midespliegue-6c9ddb5cf-twnfw        1/1     Running   0           51m
midespliegue-6c9ddb5cf-vss76        1/1     Running   0           51m
midespliegue-6c9ddb5cf-z7kvw        1/1     Running   0           18s
sergi@ubuntu:~$
```

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-escalar a 10).

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

6. 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
#Características 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:

```
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ kubectl apply -f "deployment.yaml"
deployment.apps/midespliegue created

```

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:

```

sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ kubectl apply -f "service.yaml"
service/midespliegue-http created
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ kubectl get services
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP  PORT(S)    AGE
kubernetes          ClusterIP   10.96.0.1        <none>       443/TCP    3d1h
midespliegue-http   ClusterIP   10.105.249.52    <none>       5000/TCP    8s
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ kubectl get deployments
NAME          READY  UP-TO-DATE  AVAILABLE  AGE
midespliegue  5/5    5           5          5m13s
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$

```

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
  #Mínimo y máximo 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”**.

```
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ minikube service midespliegue-http
-----
| NAMESPACE | NAME           | TARGET PORT | URL                               |
|-----|-----|-----|-----|
| default   | midespliegue-http | 5000        | http://192.168.49.2:31848 |
|-----|-----|-----|-----|
🔗 Opening service default/midespliegue-http in default browser...
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$ kubectl get pods
NAME                                READY STATUS RESTARTS AGE
midespliegue-7ddb9fd5f5-9n48f      1/1 Running 1 18h
midespliegue-7ddb9fd5f5-ds8dt      1/1 Running 0 59s
midespliegue-7ddb9fd5f5-p92jl      1/1 Running 0 59s
midespliegue-7ddb9fd5f5-rw5tg      1/1 Running 1 18h
midespliegue-7ddb9fd5f5-zqxsx      1/1 Running 1 18h
sergi@ubuntu:~/Desktop/KubernetesUD08/CasoPractico1$
```

7. 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
```

8. BIBLIOGRAFÍA

- [1] Kubernetes <https://kubernetes.io/>
- [2] Kubernetes docs <https://kubernetes.io/docs/home/>
- [3] MiniKube <https://minikube.sigs.k8s.io/docs/>