Samuel García Sánchez
Andrés Álvarez de Cienfuegos Cremades

PRÁCTICA CREATIVA 2

Despliegue de una aplicación escalable.

BLOQUE 1: Despliegue de la aplicación en máquina virtual pesada

Creamos un código de python con la ayuda de la interfaz de comandos *gcloud* y la librería de python *google.cloud.compute_v1*.
El código sigue el siguiente ciclo:

1. Init: Inicia la variable de entorno *GOOGLE_APPLICATION_CREDENTIALS* a un fichero *key.json* que nos proporciona la consola de GCloud. Esta variable será utilizada por la CLI.
2. Create: Crea la máquina con las siguientes especificaciones (se pueden especificar cómodamente en forma de parámetros en la línea de comandos o como constantes al comienzo del código fuente):
    a. Instance name: “instance-1”.
    b. Project ID: “cdps-creative-2”.
    c. Zone: “europe-west1-b”.
    d. Machine type: “e2-medium”.
    e. Network-tier: “PREMIUM” para agilizar la descarga del código fuente de la aplicación.
    f. Tags: “http-server,https-server” ya que se va a desplegar una web.
    g. Imagen: Debian 10.
3. List Instances: Listar las instancias de un proyecto creadas en una zona.
4. Deploy: Mediante el comando *gcloud compute ssh* ejecuta en la máquina las siguientes acciones:
    a. Actualización de los repositorios de Debian.
    b. Instalación de git y pip.
    c. Borrado del código fuente (si se ha clonado anteriormente).
    d. Clonado del código fuente del repositorio de Github.
    e. Instalación de las dependencias.
    f. Modificación del código fuente con el comando *sed* para que se presente el número del grupo como título de la página.
    g. Arrancado de la aplicación usando el comando *setsid -f* para que arranque en el background y enviando el output a un fichero. Esto último no solo servirá para debuggear un posible fallo, sino que liberará el comando y permitirá al script salir.

Tras ejecutar el ciclo de comandos, podemos entrar en la IP asignada por GCloud y el puerto especificado en el script (9080) y accederemos a la aplicación.

BLOQUE 2: Despliegue de una aplicación monolítica usando docker

Contenido de Dockerfile

```
FROM python:3.7.7-slim
ENV GROUP_NUMBER=43
EXPOSE 9080
RUN apt update
RUN apt install git -y
RUN git clone https://github.com/CDPS-ETSIT/practica_creativa2.git
WORKDIR /practica_creativa2/bookinfo/src/productpage
RUN pip3 install -r requirements.txt
RUN sed -i "/block\ title/ s/}.*{/}$GROUP_NUMBER{/"
./templates/productpage.html
CMD [ "python", "./productpage_monolith.py", "9080" ]
```

Creación del contenedor

Primero descargamos la imagen de python (versión 3.7.7 para evitar problemas de dependencias surgidos) del repositorio de Docker:

```
$ docker pull python:3.7.7-slim
```

Compilamos nuestra imagen partiendo de la imagen que acabamos de descargar y la armamos según hemos indicado en el dockerfile. Esta imagen se llamará según nuestro número de grupo y el nombre del servicio:

```
$ docker build -t 43/productpage_monolith .
```

Listamos las imágenes con `$ sudo docker images`. Comprobamos que la imagen se ha creado correctamente.

Finalmente, creamos y arrancamos el contenedor a partir de nuestra imagen, llamada 43/productpage_monolith. Mapeamos el puerto 80 del host con el 9080 del contenedor para poder acceder desde el host a su contenido.

```
$ docker run -d --name 43-productpage_monolith -p 80:9080 \
                43/productpage_monolith
```

Acceso al contenedor

Accedemos a través del navegador de la siguiente forma: http://localhost ya que hemos mapeado el puerto 80 de nuestro host con el 9080 del contenedor.

BLOQUE 3: Segmentación de una aplicación monolítica en microservicios utilizando docker-compose.

En esta parte se han implementado los servicios de forma independiente mediante docker-compose. Para ello, se ha creado un dockerfile para cada servicio, y mediante docker compose (que no es más que un wrapper de docker) se han construido (usando los ficheros dockerfile creados anteriormente), levantado y realizado las configuraciones necesarias en los contenedores (mapeo de puertos, variables de entorno, conexión entre ellos...).

Diferencias con aplicación monolítica

Esta implementación es ventajosa respecto a la anterior por varios motivos. En primer lugar permite una mejor organización al ser los servicios independientes ya que es posible organizarse en equipos o por partes. Además, el hecho de que estén segmentados hace que el código sea menos extenso, lo que implica por un lado que sea más entendible y por otro que los servicios sean más ligeros y arranquen más rápido. También, al ser independientes, si un servicio cae no afectaría al resto. En resumen, la segmentación de la aplicación supone mejoras en el despliegue, mantenimiento y testeo (filosofía DevOps), permitiendo el desarrollo continuo de aplicaciones más grandes y complejas.

Como todo, también cabe recalcar algunos aspectos negativos. El primero es que hay que implementar la comunicación entre los servicios, algo que no es trivial y puede ser difícil de testear. Además, de esta forma se produce un mayor consumo de memoria, estamos cambiando N instancias de una aplicación monolítica por NxM instancias de servicios, y dado que cada servicio está aislado conlleva un aumento de los recursos consumidos. En este caso no es tan grande el aumento de recursos al tratarse de virtualización ligera, pero si cada servicio corriese en una VM independiente sería algo muy a tener en cuenta. Pero, volviendo al principio, el problema principal sería la comunicación entre servicios, ya que hay que tener especial cuidado con las peticiones que se realizan entre servicios.

Despliegue

El despliegue lo hemos realizado ejecutando la siguiente instrucción dentro del directorio docker-compose: `$ docker-compose up -d` (si también queremos construir las imágenes, añadiremos la flag `--build`).

BLOQUE 4: Despliegue de una aplicación basada en microservicios utilizando Kubernetes

Utilizaremos el fichero docker-compose.yml realizado en el apartado anterior para automatizar la construcción de las imágenes docker.

```
$ docker-compose build
```

Una vez construidas las imágenes, procedemos a cargarlas para que kubernetes pueda acceder a ellas. Se ha optado por cargarlas directamente a minikube mediante el comando `$ minikube image load` (también se podrían subir a docker-hub, al repositorio del proyecto de GCloud o al repositorio que general el add-on de minikube).

Procedemos a desplegar los contenedores mediante el uso de ficheros de configuración yaml:

- Productpage:
  - Servicio con nombre productapp que expone el puerto 80, al que accederemos a la web.
  - Deployment con una única réplica usando la imagen 43/productpage.
- Details:
  - Servicio exponiendo el puerto 9080 al que accede productpage.
  - Deployment con 3 réplicas y usando la imagen 43/details:v1.
- Reviews:
  - Servicio exponiendo el puerto 9080 al que accede productpage. En el selector de las especificaciones (spec/selector) se selecciona la app con nombre reviews y la versión que se quiera mostrar (v1, v2 o v3).
  - Se despliega un deployment diferente por cada versión (usando la imagen de la versión correspondiente, 43/reviews:vx). Al tener las 3 versiones desplegadas, podremos cambiar entre ellas sin que el servicio pare en ningún momento.
- Ratings:
  - Servicio exponiendo el puerto 9080 al que accede productpage.
  - Deployment con 2 réplicas y usando la imagen 43/ratings:v1.

Aplicamos las configuraciones (cuyos ficheros se encuentran en la carpeta kubernetes/) con el siguiente comando:

```
$ kubectl apply -f kubernetes
```

Accedemos al servicio de productapp mediante la url facilitada por el comando:

```
$ minikube service productapp --url
```

Debilidades de la arquitectura

Fiabilidad

De las implementaciones descritas anteriormente la única que ofrece la replicación de los servicios es kubernetes. Esto significa que en el caso de pérdida de un POD, el servicio seguirá corriendo. Además kubernetes nos ofrece la posibilidad de monitorizar los servicios y notificarnos rápidamente en caso de fallo.

Cabe resaltar que no todos los servicios tienen réplicas, en especial cabe resaltar la ausencia de réplicas en la página principal (productpage). Si falla el POD de esta aplicación, se perderá el servicio completo.

Al no disponer de bases de datos, la información de los productos podría perderse en el caso de fallo de la aplicación (ya que al volver a arrancar la aplicación volverían a los valores por defecto). En este caso, estas informaciones son estáticas y no es necesario porque siempre es igual, pero en el caso de que estos datos pudieran cambiar, sería necesario desplegar contenedores con bases de datos.

Escalabilidad

La única opción (de manera sencilla) que nos permite escalar los servicios es kubernetes, creando o parando réplicas conforme el número de accesos aumenta o disminuye. Si es cierto que, como se ha comentado en el Bloque 3, con docker-compose ya obtenemos la segmentación de la aplicación y se podrían desplegar balanceadores de carga y proxies reversos que permitieran la escalabilidad del servicio, pero este sistema es mucho más complejo que el despliegue con kubernetes.

Conforme aumenten las entradas de productos, las aplicaciones se cargarán de datos y bajará su rendimiento. Para solventar este problema, se podrían implementar bases de datos que almacenen esta información. Obviamente, estas bases de datos, se deberían desplegar en sus correspondientes PODs con réplicas y backups.