Mateo Sarria Franco de Sarabia
Rafael Bueno Castro
Jacobo España-Heredia Beteta

Práctica creativa 2

Despliegue de una aplicación escalable

Los bloques 1, 2 y 3 se realizan desde una instancia VM en Google Cloud. El bloque 4 se lleva a cabo desde la consola de GKE.

1. Despliegue de la aplicación en máquina virtual pesada

Para el primer despliegue, ejecutamos un script en Python *bloque1.py* y accedemos a la dirección IP externa de la máquina virtual (*<ip_externa>:9080/productpage*).

El script realiza las siguientes acciones: en primer lugar, clona el repositorio de la práctica en GitHub. A continuación, realiza las modificaciones necesarias en *requirements.txt* e instala las dependencias especificadas en este archivo. Después, cambia el título de la página al número del grupo guardado en *GROUP_NUMBER* y, por último, lanza la aplicación web en el puerto 9080 con el script *productpage_monolith.py*.

El uso de scripts para la automatización se caracteriza por su alta portabilidad y facilidad de mantenimiento, lo que la convierte en una práctica eficiente. Sin embargo, cuando se trata de escalabilidad, el modelo monolítico adoptado en esta sección puede resultar limitado, especialmente en escenarios con grandes volúmenes de carga o despliegues a gran escala. Para mitigar estos inconvenientes, sería recomendable incorporar mecanismos de verificación del estado del sistema y optimizar las herramientas de monitoreo. Esto permitiría asegurar una mayor disponibilidad y mejorar el rendimiento general de la aplicación.

2. Despliegue de una aplicación monolítica usando docker

En el segundo bloque, desplegamos la misma aplicación monolítica utilizando Docker. Para realizar el despliegue, ejecutamos el script *bloque2.py* y accedemos a la dirección IP externa de la máquina virtual (*<ip_externa>:9080/productpage*).

El script realiza las siguientes acciones: en primer lugar, creamos la imagen de Docker localmente a partir del archivo *Dockerfile*. A continuación, arranca el contenedor en el puerto 9080 y asigna a *GROUP_ENV* el valor 37. El nombre de la imagen es *product-page/g37* y el del contenedor es *product-page-g37*.

El archivo *Dockerfile* contiene las especificaciones para la creación de la imagen, entre las que se encuentran: la base (*python:3.7.7-slim*), la instalación de Python y pip3, la exposición del puerto 9080, la definición de la variable de entorno *GROUP_NUMBER*, el clonado del repositorio de la práctica, la copia y ejecución del programa *rename.py* (que modifica los archivos *requirements.txt* e *index.html* de productpage) y la instalación de las dependencias especificadas en *requirements.txt*. Una vez creado el contenedor, se ejecuta el script *productpage_monolith.py* en el puerto 9080, permitiendo el acceso a la aplicación.

Para deshacer el escenario, ejecutamos el script *delete.py*, que elimina la imagen y el contenedor creados.

Aunque este enfoque presenta varias ventajas, también se identifican algunas debilidades en la implementación actual, principalmente vinculadas a la fiabilidad y escalabilidad. Entre los aspectos más destacados están la gestión y resolución de las dependencias especificadas en el archivo *requirements.txt*, así como la elección de una arquitectura monolítica frente a una basada en microservicios, que podría ofrecer mayores beneficios en modularidad y crecimiento.

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

La migración de una arquitectura monolítica a un modelo basado en microservicios utilizando Docker y Docker Compose representa un cambio profundo en la manera de desarrollar y desplegar aplicaciones. Este enfoque se centra en dividir la aplicación en múltiples servicios independientes, cada uno encargado de una función específica.

La adopción de microservicios ofrece importantes beneficios, como mayor flexibilidad, escalabilidad y facilidad de mantenimiento. Además, mejora la gestión del sistema, el desarrollo futuro y la interoperabilidad entre servicios.

Para realizar el despliegue, ejecutamos el script *bloque3.py* seguido de la versión (v1, v2 o v3) que deseamos utilizar y accedemos a la dirección IP externa de la máquina virtual (*<ip_externa>:9080/productpage*).

El script realiza las siguientes acciones: primero, clona el repositorio de la práctica localmente y copia los archivos que deben ser utilizados por los contenedores en sus respectivas carpetas. A continuación, desde el directorio correspondiente, ejecuta el comando de compilación y ejecución de *reviews*. Posteriormente, modifica el archivo *docker-compose.yaml* con la versión deseada de la web y, finalmente, construye las imágenes y lanza el docker-compose.

La orquestación de los servicios independientes se lleva a cabo con Docker Compose, que permite definir y gestionar cada servicio a través del archivo *docker-compose.yaml*. En este archivo se especifican los pasos para construir las imágenes Docker, configurar los contenedores y establecer las variables de entorno necesarias para el funcionamiento de cada componente.

Para deshacer el escenario, ejecutamos el script *delete.py*, que elimina las imágenes y contenedores creados.

Sin embargo, este enfoque no está exento de desafíos. La administración de varios microservicios introduce una mayor complejidad en la gestión de redes y comunicaciones entre ellos. Para superar estas dificultades, se recomienda adoptar herramientas de orquestación más avanzadas, como Kubernetes, que ofrecen una gestión más eficiente y automatizada de los microservicios, incluyendo escalado dinámico y balanceo de carga.

Mateo Sarria Franco de Sarabia
Rafael Bueno Castro
Jacobo España-Heredia Beteta

4. Despliegue de una aplicación basada en microservicios utilizando Kubernetes

Para este último bloque, creamos un clúster con las características deseadas. Una vez creado, desde la consola de GKE, lanzamos el script *bloque4.py* seguido de la versión elegida (v1, v2 o v3) y accedemos a través de la IP externa del balanceador de carga de *productpage* obtenida con el comando *kubectl get services*.

El script automatiza la creación y el lanzamiento de los servicios con Kubernetes. Estos servicios están definidos en archivos *.yaml* (uno para cada servicio, además de uno para la definición del servicio *reviews* y tres para su *deployment*, en función de la versión). Las imágenes de los servicios se encuentran en un repositorio de Docker Hub (*mateosarria/pc2:<servicio>*).

Para deshacer el escenario, ejecutamos el script *delete.py*, que elimina todos los pods y servicios de Kubernetes creados.

Uno de los aspectos más destacados de Kubernetes es su capacidad para replicar nodos, lo que incrementa significativamente la fiabilidad y disponibilidad de los servicios, especialmente en caso de fallos o interrupciones.

Además, dentro de esta capacidad de replicación, Kubernetes realiza una distribución automática de la carga entre todos los pods disponibles, optimizando el uso de recursos y garantizando un rendimiento más equilibrado y eficiente de las aplicaciones desplegadas. Esta funcionalidad no solo mejora la resiliencia del sistema, sino que también facilita la escalabilidad horizontal de los servicios.