MEMORIA PRÁCTICA CREATIVA 2:

Despliegue de una aplicación escalable

GRUPO 41:
Andrés Herencia López-Menchero
Patricia Belén Fernández Sastre
Fernando Herrera Pozo

ÍNDICE

1. Introducción 3

2. Despliegue de la aplicación en máquina virtual pesada 3

3. Despliegue de una aplicación monolítica usando docker 4

4. Segmentación de una aplicación monolítica en microservicios utilizando docker-compose 5

Conclusiones 7

Anexos 9

1. Introducción

En esta práctica hemos realizado la creación de un escenario completo para poder desplegar una aplicación usando tres tecnologías diferentes: máquina virtual pesada, docker y microservicios en docker-compose.

2. Despliegue de la aplicación en máquina virtual pesada

En esta primera parte nuestro objetivo es desplegar la aplicación como si fuera un **monolito**, utilizando técnicas tradicionales.

Hemos optado por la opción de utilizar la **infraestructura de Google Cloud** y desplegar allí las máquinas virtuales, ya que nos ha parecido una forma más sencilla puesto que así no teníamos que configurar los servidores s1 a s3.

Hemos creado un script de Python llamado ***script1.py*** [1] que hace las siguientes operaciones:

1) **Instalar *pip3***, para poder posteriormente instalar los *requirements.txt*, las dependencias que necesita nuestra aplicación.

2) **Clonar la práctica** desde GitHub (comando *git*).

3) Definir la variable de entorno (con la librería *os* y el método *environ*) de ***GROUP_NUMBER***, con valor asignado a nuestro grupo, 41. Puesto que este script se ejecuta solo para nuestro grupo, no hemos visto la necesidad de introducirlo por consola como argumento.

4) **Editar el fichero** (con *with open* en modo lectura y escritura) para cambiar el título de tal manera que ponga nuestro número de grupo, con la variable de entorno ya definida.

5) **Instalar dependencias con *pip3***, tal como se ha indicado anteriormente.

6) Llamar al subscript ***productpage_monolith.py*** con el argumento **9080**, especificando el puerto. Recordar que el script principal que ejecutamos no es el que coge el puerto, si no que este ya ha sido especificado en el propio script y pasado como parámetro al monolith.

3. Despliegue de una aplicación monolítica usando docker

En esta parte, vamos a llevar a cabo el despliegue utilizando la virtualización ligera de **Docker**.

Para ello, lo que hemos hecho ha sido crear un fichero Dockerfile [2], que será en el que nos apoyaremos para construir nuestro contenedor y posteriormente desplegarlo. Hemos llevado a cabo los siguientes pasos:

1) **Hemos especificado una imagen de Python**, la *3.7.7-slim*. Es slim para que ocupe menos espacio y sea más eficiente. También podría haberse hecho con *Alpine*, pero tendríamos que haber instalado las dependencias aparte.
2) Posteriormente, **exponemos el puerto** donde queremos lanzar la aplicación, el 9080.
3) Al igual que en el script, **instalamos git y clonamos el proyecto**.
4) **Cambiamos el directorio de trabajo** al adecuado para instalar los *requirements.txt* y modificar el título.
5) Definimos la variable de entorno **y la exportamos a un subscript** [3] que lo que haga sea modificar el título con el valor de ***GROUP_NUMBER*** de nuestra variable de entorno. Este proceso se podría hacer con sed, escapando los carácteres adecuados.
6) **Instalamos los *requirements.txt* con pip**.
7) Como en cualquier Dockerfile, debemos de hacer que el contenedor ejecute con ***ENTRYPOINT*** o ***CMD*** una orden para lanzar la aplicación.

Para construir el contenedor, debemos de ejecutar la siguiente orden:

```
docker build -t g41/productpage .
```

Donde el argumento "-t" especifica el tag de nuestro contenedor, para que podamos llamarlo con run posteriormente y ponemos ".", especificando que estamos construyéndolo sobre el Dockerfile de nuestro directorio de trabajo actual.

Por último, desplegamos el contenedor con la siguiente orden:

```
sudo docker run --name g41-productpage -p 9080:9080 -e GROUP_NUMBER=41 -d
g41/productpage
```

- -- name: especifica el nombre de la imagen, lo hacemos tal y como se especifica en el enunciado: <número de grupo>-<nombre de servicio>.
- - p: sirve para mapear el puerto interno del propio “docker” al puerto de nuestro ordenador (host). La aplicación internamente se despliega en el puerto 9080 de nuestro contenedor, que es al que redireccionaremos en nuestra máquina host.
- - e: especifica la variable de entorno. La usamos para especificar GROUP_NUMBER a 41.
- - d: usamos el modo “detatched” en nuestra run. Opcional.
- Finalmente, especificamos qué imagen arranca, en nuestro caso, a la que acabamos de hacer build “g41/productpage”.

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

En esta última parte, la tecnología que hemos empleado para desplegar la aplicación ha sido **docker-compose**. Lo que hemos hecho es segmentar la aplicación en dos microservicios (Product Page y Details), para que puedan funcionar de forma independiente, y hemos añadido dos nuevos: Ratings y Reviews.

Para llevar a cabo esta tarea, hemos llevado a cabo varios pasos:

1) Hemos escrito un archivo de tipo **Dockerfile** para cada microservicio [4][5][6], con las acciones requeridas para cada uno, y también construido su imagen correspondiente, mediante los comandos **docker build -t g41/XX**. Se puede consultar en anexos.

    ➢ Servicio **Product Page**
    ➢ Servicio **Details**
    ➢ Servicio **Reviews**
    ➢ Servicio **Ratings**

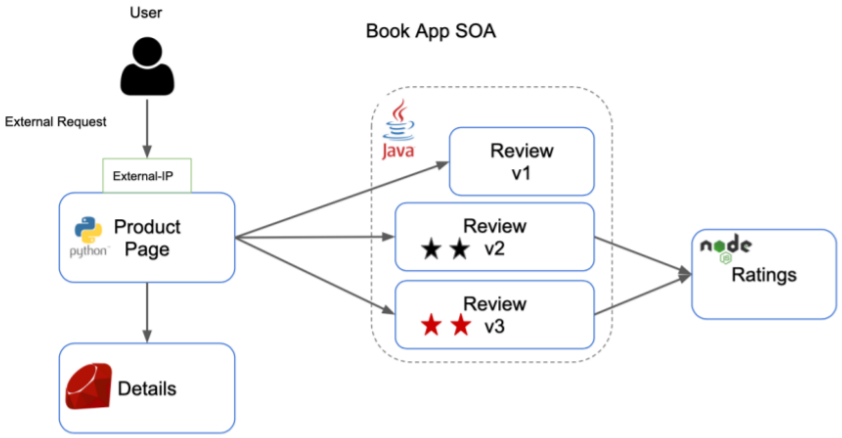
2) Una vez hemos creado los 4 dockerfiles con sus correspondientes imágenes, lo que hacemos es crear el fichero docker-compose **para unir** estos microservicios.

    El contenido del **docker-compose** es el siguiente:

```yaml
version: '3'
services:

  product-page:
    image: g41/product-page
    ports:
      - "9080:9080"
  details:
    image: g41/details
    ports:
      - "9080"
    environment:
      - SERVICE_VERSION=v2
      - ENABLE_EXTERNAL_BOOK_SERVICE=true
  reviews:
    image: g41/reviews
    ports:
      - "9080"
    environment:
      - ENABLE_RATINGS=true
      - SERVICE_VERSION=v2
      - STAR_COLOR=red
  ratings:
    image: g41/ratings
    ports:
      - "9080"
    environment:
      - SERVICE_VERSION=v2
```

Para crearlo, nos hemos basado en el siguiente esquema:



Podemos observar que el servicio Product Page está unido a Details y a las 3 versiones de Reviews, y que las versiones 2 y 3 de Reviews están unidas directamente con el servicio Ratings. El usuario se conecta a través de la dirección IP externa al servicio Product Page.

Para especificar los servicios usamos la etiqueta ***services***. Cada servicio obtiene una imagen del contenedor con ***image***, y se especifica los puertos en los que se ejecuta con ***ports***. Las variables de entorno, se especifican con ***environment*** con un valor por defecto. Este valor por defecto puede ser a nuestra elección y se podría cambiar con un fichero *envfile*. De cualquier manera, son arbitrarios pero dinámicos. Recordar la importancia de la identación (y los espacios) en ficheros del tipo .yaml.

Una vez creado, y antes de desplegar nuestro docker-compose, tenemos que construir la imagen de cada contenedor para que el docker-compose las encuentre y las ejecute al hacer docker-compose up.

Siguiendo las especificaciones del enunciado, compilamos y empaquetamos los ficheros necesarios ejecutando, dentro de la ruta *src/reviews/reviews-wlpcfg*, el siguiente comando:

```
docker run --rm -u root -v "$(pwd)":/home/gradle/project -w
/home/gradle/project gradle:4.8.1 gradle clean build
```

Para cada imagen (ya una vez hecho el build como se especificó en esta misma sección), obtenemos que al hacer **docker-compose up** nuestra aplicación funciona, desplegada por microservicios. Consultamos http://localhost:9080 para comprobarlo, o bien la desplegamos en Google Cloud y usamos su IP externa.

Las **diferencias** de emplear docker-compose frente a docker son que podemos ejecutar múltiples contenedores a la vez, permitiendo simplificar el uso de docker, y que en vez de utilizar una serie inmemorizable de comandos y scripts, podemos mediante archivos **YAML** realizar tareas de forma más sencilla.

Conclusiones

Cada tecnología tiene sus puntos fuertes y débiles. Por un lado, el despliegue de aplicaciones sobre **máquinas virtuales pesadas** a priori es **fiable** pero tiene el problema, como su nombre indica, de que las aplicaciones se despliegan de manera pesada, hace que el despliegue de las aplicaciones sea **más lento, menos eficiente y costoso**.

Para solucionar esto, se empezaron a usar contenedores, **Dockers**, que lo que hacen es ejecutar las aplicaciones sobre contenedores ligeros, algo así como máquinas virtuales que se ejecutan sobre el **S.O. del host** en vez del de la propia imagen, aligerando el despliegue de las aplicaciones. Aunque fiables, su **escalabilidad es limitada**, ya que hay que desplegar uno por uno cada uno de los contenedores para obtener servicios, limitando la modularidad y prácticamente obligando **a usar aplicaciones monolíticas**.

Para solucionar esto, tenemos **docker-compose**, que lo que hace es desplegar los contenedores a la vez, en vez de uno a uno. Esto se consigue gracias a la unión de los contenedores en un fichero de configuración escrito en yaml, lo cual viene a solucionar el problema de escalabilidad e incluso usabilidad de los contenedores cuando tenemos más de un servicio, permitiendo fácilmente desplegar los llamados **microservicios**. Sin embargo, docker-compose tiene un problema a la hora de seguir escalando, pues **cuando tenemos más de un servidor, se vuelve tedioso**.

Así es como llegamos a la última solución: **Kubernetes**, un sistema de orquestación que prácticamente se convierte en el estándar en la industria y que permite una **gran escalabilidad**. Este servicio está disponible en la mayoría de plataformas Cloud (tales como GCP, AWS, Azure). El problema es la **fiabilidad**, que puede ser más limitada en algunos casos que Docker y Docker-compose y **hay que tener claro la comunicación entre los distintos procesos**.

Sea como sea, queda claro que estas tecnologías son útiles en un espacio de DevOps y ayudan a mantener una fiabilidad, cohesión y comunicación entre todos los servicios que podemos desplegar en la nube.

Anexos

[1] Script de Python para desplegar el monolito sobre MV pesada en GCP

```
import os
import sys
from subprocess import call

path = "practica_creativa2/bookinfo/src/productpage"

call(["sudo", "apt", "install", "-y", "python3-pip"])
call(["sudo", "apt-get", "install", "-y", "git"])
call(["git", "clone",
"https://github.com/CDPS-ETSIT/practica_creativa2.git"])

os.environ["GROUP_NUMBER"] = "41"
print(os.environ["GROUP_NUMBER"])

with open("{}/templates/productpage.html".format(path), "r") as f:
    data = f.readlines()

data[33] = """\n<a class="navbar-brand" href="#">BookInfo Sample. Grupo
{}</a>\n""".format(os.environ["GROUP_NUMBER"])
print(data)

with open("{}/templates/productpage.html".format(path), "w") as f:
    f.writelines(data)

call(["pip3", "install", "-r", "{}/requirements.txt".format(path), "-v"])

call(["python3", "{}/productpage_monolith.py".format(path), "9080"])
```

[2] Dockerfile que despliega la aplicación monolítica

```
FROM python:3.7.7-slim

EXPOSE 9080

RUN apt-get update
RUN apt-get install -y git
```

```
RUN git clone https://github.com/CDPS-ETSIT/practica_creativa2.git

WORKDIR practica_creativa2/bookinfo/src/productpage

ENV GROUP_NUMBER=41
COPY mod_txt.py .
RUN python3 mod_txt.py ${GROUP_NUMBER}

RUN pip install -r requirements.txt && \
    pip install urllib3==1.24.1 jsonschema==2.6.0 && \
    pip install --upgrade requests

ENTRYPOINT python3 productpage_monolith.py 9080
```

[3] Sub script para cambiar el título

```
import sys
from subprocess import call

with open("templates/productpage.html", "r") as f:
    data = f.readlines()

data[33] = """\n<a class="navbar-brand" href="#">BookInfo Sample. Grupo
{}</a>\n""".format(sys.argv[1])
print(data)

with open("templates/productpage.html", "w") as f:
    f.writelines(data)
```

[4] Dockerfile para el servicio Product Page

```
FROM python:3.7.7-slim

RUN apt-get update
RUN apt-get install -y python3-pip python3-dev

COPY . .

WORKDIR /practica_creativa2/bookinfo/src/productpage
```

```
EXPOSE 9080

RUN pip install -r requirements.txt

CMD ["python3", "productpage.py", "9080"]
```

[5] Dockerfile para el servicio Details

```
FROM ruby:2.7.1-slim
COPY practica_creativa2/bookinfo/src/details /opt/microservices/
WORKDIR /opt/microservices/
EXPOSE 9080
CMD ["ruby", "details.rb", "9080"]
```

[6] Dockerfile para el servicio Ratings

```
FROM node:12.18.1-slim

RUN apt-get update
RUN apt-get install -y git

COPY /practica_creativa2/bookinfo/src/ratings/package.json
/opt/microsrevices/
COPY /practica_creativa2/bookinfo/src/ratings/ratings.js
/opt/microsrevices/

WORKDIR /opt/microsrevices/

RUN npm install

EXPOSE 9080

CMD ["node", "ratings.js", "9080"]
```