CDPS
PRÁCTICA CREATIVA 2

DESPLIEGUE DE UNA APLICACIÓN ESCALABLE

G43:
Víctor Nieto Palacios
David Fuentes Martín
Pablo de la Cruz Gómez

1. Despliegue de la aplicación en máquina virtual pesada (2 puntos)

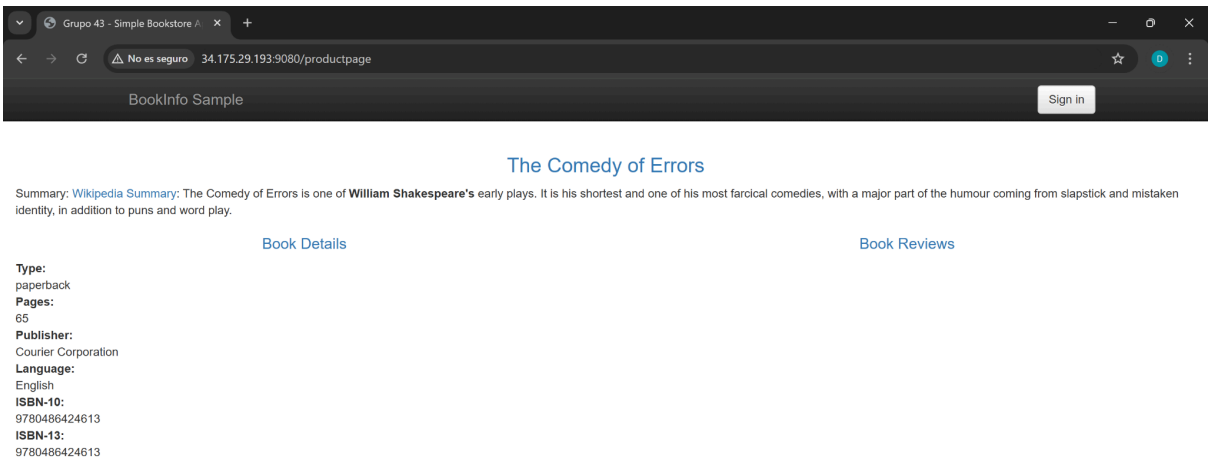
Hemos decidido programar el script que se encarga de instalar la aplicación en una VM pesada alojada en la infraestructura de google cloud.
Comenzamos creando la máquina virtual en la interfaz de Google Cloud. Instalaremos un entorno Debian como realizamos en la práctica de Docker anteriormente en la asignatura. Una vez encendida la máquina y conectándonos al terminal, debemos ejecutar los comandos necesarios para actualizar e instalar las aplicaciones necesarias (python, git, docker…).

```
sudo apt-get update
sudo apt-get install -y git
sudo apt-get install -y python3-pip
git clone https://github.com/dawfunes/PC2_CDPS
cd PC2_CDPS
```

Si queremos ejecutar la parte 2 aquí, también debemos instalar docker:

```
sudo apt-get install -y docker.io
```

Ahora podemos simplemente correr el comando `python3 PC2.py p1 <puerto>`, siendo este último argumento opcional para definir en qué puerto queremos correr la aplicación, si no pusiéramos ese argumento simplemente usará el default, que es 9080. Este comando correrá el script PC2.py con el argumento p1, que hará que se corra la parte 1 de la práctica. Lo hace con el uso de la biblioteca que hemos creado llamada lib_PC2, específicamente la función init_app, con el cual clonamos el repositorio de práctica creativa 2, cambiamos el contenido del título para que contenga el nombre del grupo, cambie el contenido de requirements.txt, los instala y por último corre el fichero productpage_monolith.py para sacar el servicio en una MV pesada.



Esta estructura tiene un problema de escalabilidad claro y la necesidad de muchas máquinas pesadas para levantar servicios diferentes. Esto es facilmente mejorable con un sistema de virtualizacion ligera en contenedores.

2. Despliegue de una aplicación monolítica usando docker (2 puntos)

Para esta parte, se ejecutará el comando `py PC2.py p2` start para levantar la aplicación. Este comando, llama a la función `init_app_docker()`, que primero comprueba si están los ficheros necesarios, si no lo están los clonara del repositorio de github aportado por los profesores.

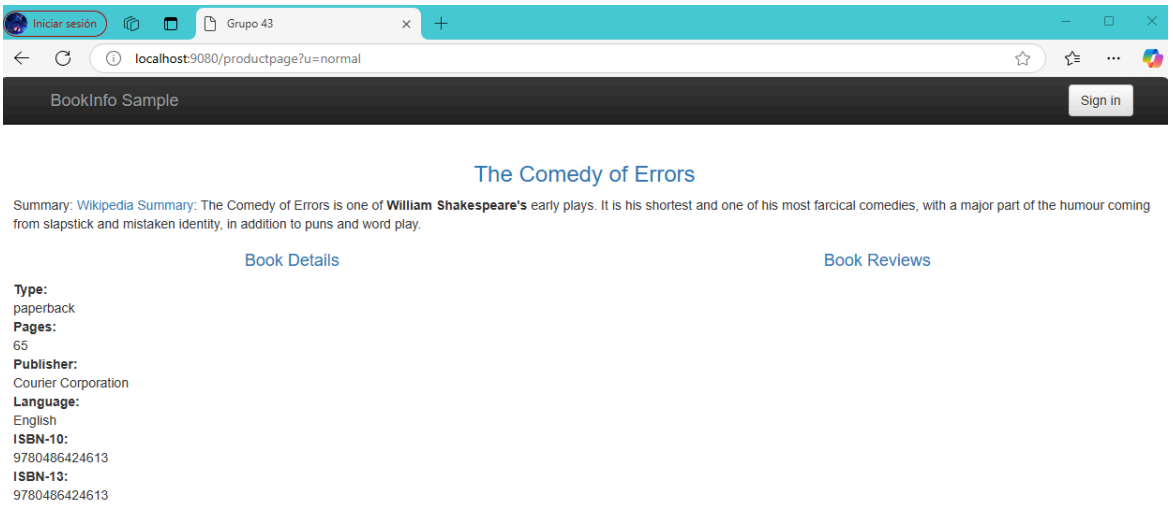
A continuación se construye la imagen de la aplicación a partir de un Dockerfile, en el que se hace lo siguiente:

1. Se utiliza *python* como imagen base
2. Se indica el puerto accesible *5080*
3. Se pasa al contenedor la variable *GROUP_NUM* con valor asignado *43*
4. Se establece el directorio de trabajo en */app*
5. Se copia el directorio */productpage* dentro del contenedor.
6. Se instala *requirements.txt*
7. Se le asigna al título de la página *Grupo <GROUP_NUM>*, cambiandolo en *templates/productpage.html*
8. Se ejecutará *productpage_monolith.py* por el puerto *5080*

Una vez construida la imagen, se crea el contenedor con nombre product-page-g43, siendo visible en nuestro puerto local 9080, conectando los dos mediante 9080:5080.

También se podrá ejecutar el comando `py PC2.py p2 destroy` para eliminar tanto la imagen como el contenedor creados.

La web queda de la siguiente forma:



El problema que tiene esta estructura es su poca escalabilidad, que tiene fácil solución utilizando Docker Compose para poder definir y gestionar múltiples contenedores y así desplegar múltiples instancias y servicios.

3. Segmentación de una aplicación monolítica en microservicios utilizando docker-compose (2 puntos)

Para esta parte hemos definido 3 ficheros Dockerfile (para Product Page, Details y Ratings) y un fichero *docker-compose-base.yml* con el que se pueden iniciar estos 3 servicios (el servicio Reviews se añadirá al fichero usando el script).

Para lanzar esta parte, ejecutaremos el archivo PC2.py de la siguiente forma:

```
py PC2.py p3 start
```

Estos parámetros llamarán a la función *init_app_docker_compose()* definida en lib_PC2.py, que hace lo siguiente:

1. Clona el repositorio de la practica_creativa2 si no está todavía
2. Pide la versión de la aplicación a ejecutar (v1, v2 o v3).
3. Hace una copia del fichero *docker-compose-base.yml* y define las variables *star_color* y *enable_ratings* en función de la versión escogida.
4. Añade el texto correspondiente al microservicio reviews al fichero *docker-compose.yml* (copia de *docker-compose-base.yml*), eligiendo las variables de entorno en función de las variables definidas en el script.
5. Ejecuta el comando pedido para compilar y ejecutar los ficheros necesarios antes de construir la imagen de reviews.
6. Construye la imagen *reviews/g43* a partir del Dockerfile en el directorio *reviews-wlpcfg*.
7. Se construye la imagen de los microservicios Product Page, Details y Ratings con el comando docker-compose build.
8. Se levantan los 4 microservicios Product Page, Details, Ratings y Reviews con el comando docker-compose up.

Una vez termina de ejecutarse el comando, podremos acceder a la web en localhost:9080, y podremos comprobar que las estrellas del servicio Reviews cambian en función de la versión, y además, el título de la página será Grupo 43.

Para destruir los contenedores, simplemente tendremos que ejecutar el comando:

```
py PC2.py p3 destroy
```

Que además, borrará el archivo *docker-compose.yml*, dejando simplemente la versión base de este.

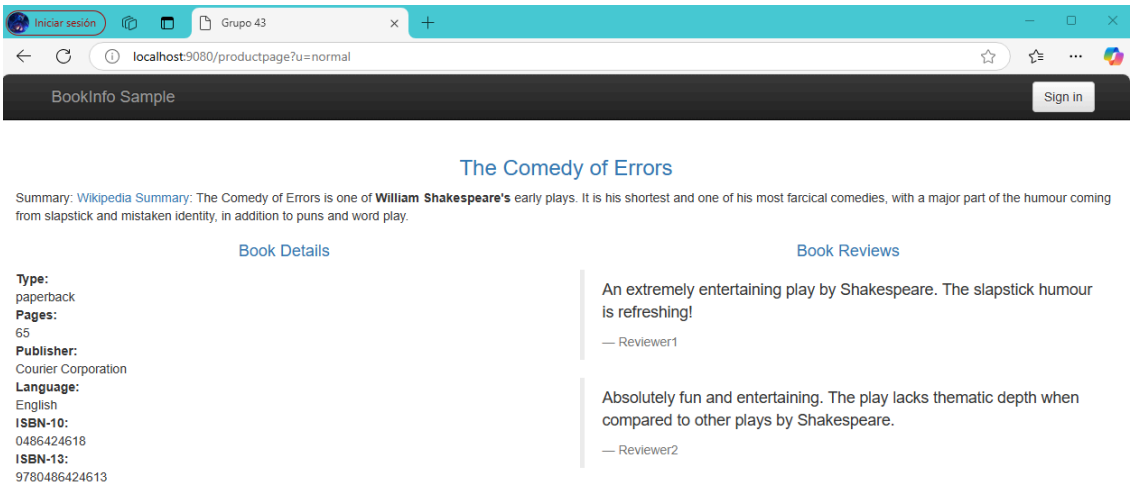
Hemos decidido usar un fichero docker-compose de base y una copia de este para lanzar los servicios, ya que así no necesitamos tener un archivo docker-compose para cada versión. Además, si quisiéramos añadir más versiones del servicio reviews, simplemente tendríamos que añadir las variables correspondientes al script, sin tener que crear otro docker-compose más.

Esto es lo primero que vemos al ejecutar el script:

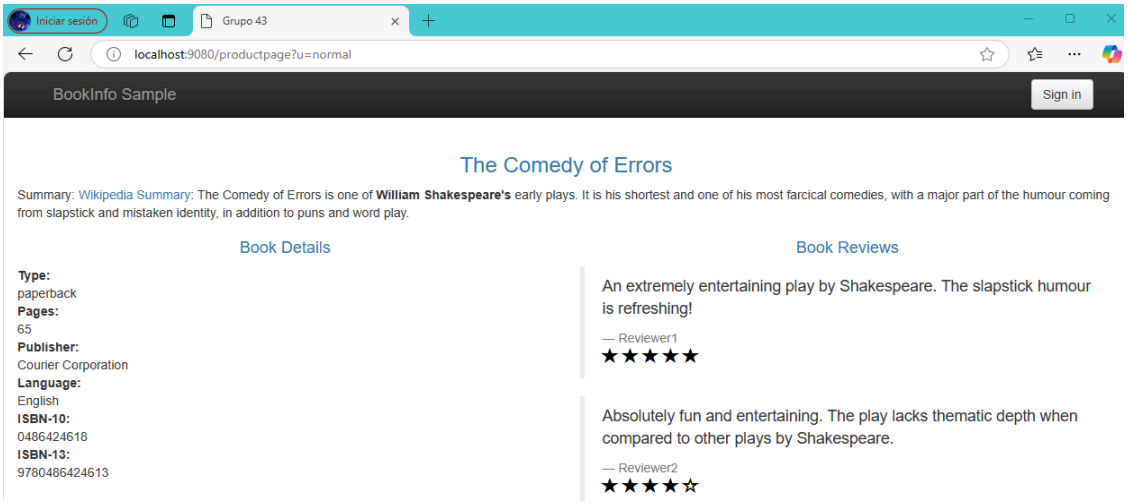
```
PS D:\Escritorio\PC2_CDPS> py PC2.py p3 start
Despliegue de la aplicación multiservicio mediante docker-compose
Repositorio ya clonado
Escoge una versión (v1, v2, v3): v1
Ejecutando la versión v1
```

Así queda la página web con las distintas versiones:

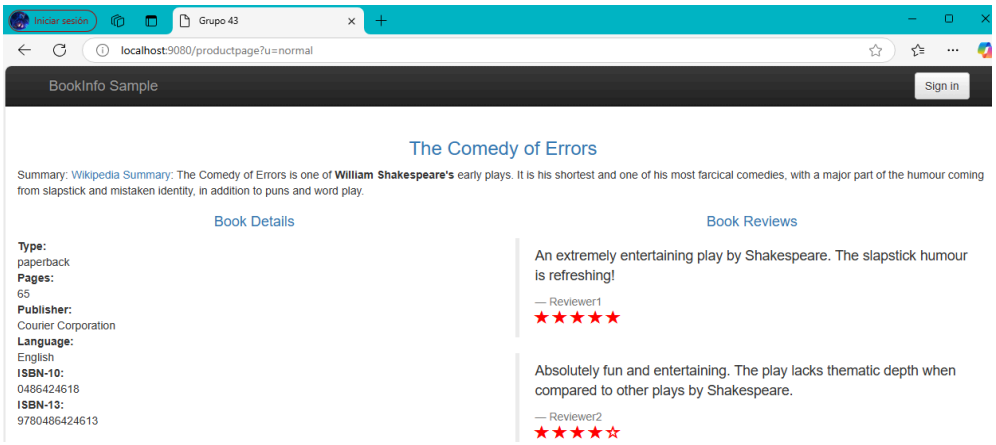
Versión 1



Versión 2



Versión 3



El problema con esta estructura es su falta de escalabilidad de manera fácil, podemos mejorarla haciendo uso de kubernetes. Lo bueno que tiene docker-compose es su facilidad para la creación de varios contenedores para la puesta a disposición de servicios.

4. Despliegue de una aplicación basada en microservicios utilizando Kubernetes (4 puntos)

Tenemos la carpeta *kubernetes* que tiene todos los ficheros *yaml* que ponen a disposición los servicios en diferentes pods. En todos tenemos el deployment y el service en un solo fichero. Reviews tiene tres ficheros, uno para cada versión. Para hacer el despliegue hemos usado minikube.

Para iniciar esta parte, una vez ya hemos iniciado minikube, usamos el siguiente comando:

```
py PC2.py p4 start
```

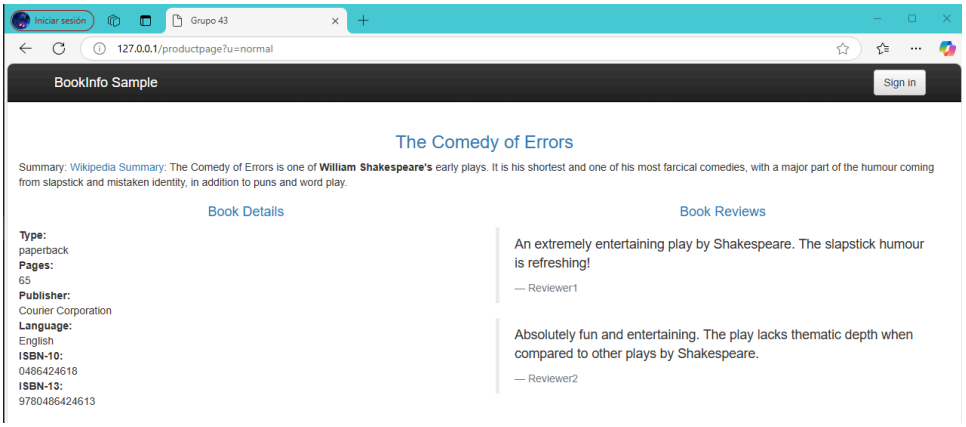
Este, pide la versión a ejecutar y aplica los archivos *yaml* correspondientes. Además ejecuta el comando *minikube tunnel,* el cual asigna una IP externa al servicio *productpage*. Para ver esta IP podemos usar el comando *kubectl get services*, y la usaremos para conectarnos a través del navegador.

```
NAME          TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
details       ClusterIP      10.106.26.169    <none>        9080/TCP         4s
kubernetes    ClusterIP      10.96.0.1        <none>        443/TCP          67s
productpage   LoadBalancer   10.105.244.29    127.0.0.1     80:31969/TCP     5s
ratings       ClusterIP      10.105.66.237    <none>        9080/TCP         4s
reviews       ClusterIP      10.104.225.51    <none>        9080/TCP         4s
```

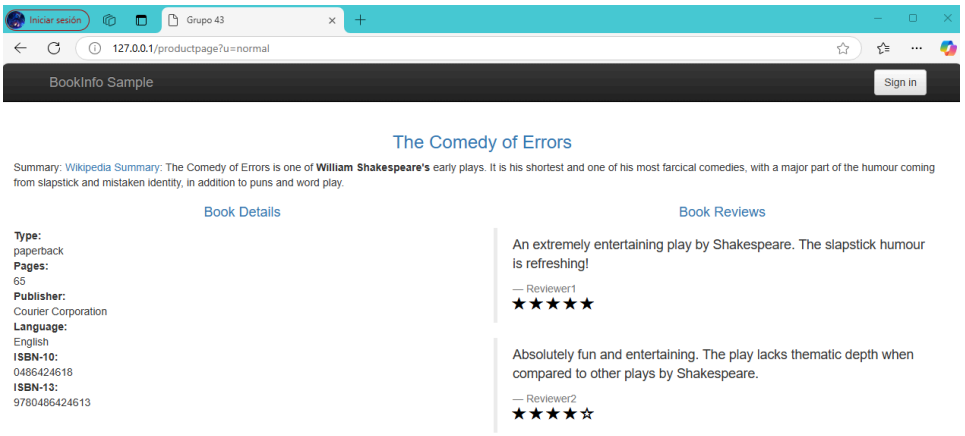
En caso de querer detener los pods y servicios basta con usar el comando:

```
py PC2.py p4 destroy
```

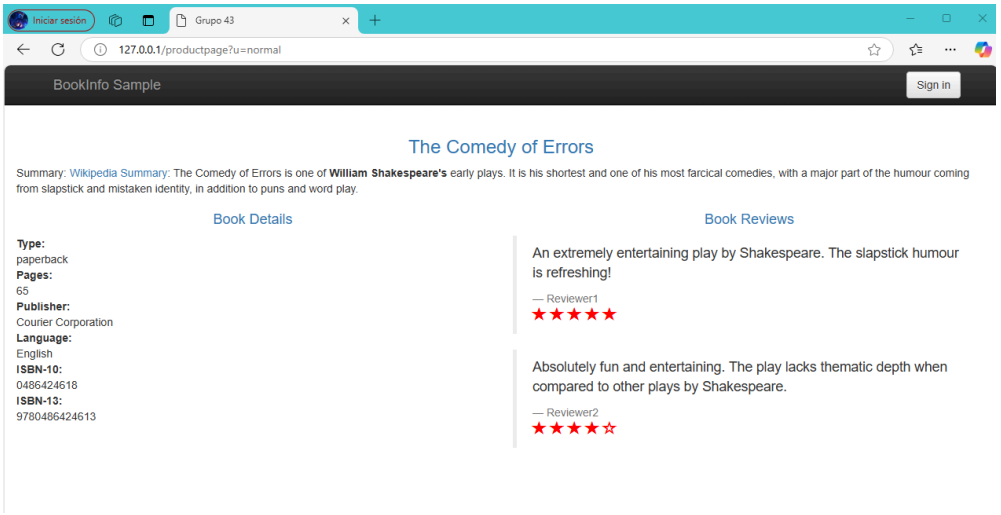
Versión 1



Versión 2



Versión 3



Un despliegue en Kubernetes permite la escalabilidad independiente de servicios, es ideal por esta razón para este escenario. Un pequeño pero, sería la complejidad inicial para gestionar la conexión entre microservicios. Si quisiéramos mejorarlo, solo podríamos mejorarlo con una aplicación externa de monitoreo (como Grafana) para estudiar el tráfico y ver si hubiera necesidad de escalabilidad de un servicio específico.