Memoria para la práctica creativa 2 CDPS

Creado por Antonio Ardura Carnicero, Sandra Cascante Morán y Martín Rodríguez Barroso Grupo 35

Se puede acceder a los mismos archivos a través de nuestro repositorio de github: https://github.com/sandracascantem/scriptsPC2

-----Part1-----DESPLIEGUE DE LA APLICACIÓN EN MÁQUINA VIRTUAL PESADA-----

La idea es desplegar la aplicación como si fuera un monolito en una máquina virtual pesada en Google Cloud. Para ello, hemos creado el script "script1.py" entregado en el zip, dentro de la carpeta "part1".

Para probarlo, hemos creado una MV en Google Cloud y abierto su consola SSH en un nuevo navegador. Además, hemos instalado git con el comando "sudo apt-get install git" para poder clonar la carpeta con los scripts. A continuación, nos cambiamos al directorio de la carpeta de esta parte (scriptsCDPS/part1).

Ejecutamos el script con el comando "python3 script1.py". Este script:

-Clona la carpeta practica_creativa2 del github de la asignatura (https://github.com/CDPS-ETSIT/practica_creativa2.git).

-Instala pip en la máquina virtual y se instalan las dependencias de requirements con pip3.

-Crea la variable de entorno "GROUP_NUMBER" que es nuestro número de grupo (35).

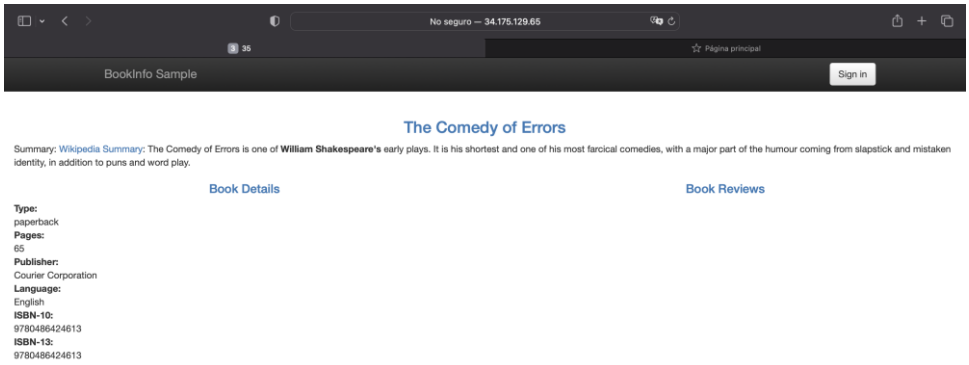
-Modifica el “productpage.html” para cambiar el título de la app por la variable de entorno.

-Pide introducir un puerto en el rango 8980-9080 para cambiarlo en el fichero de la aplicación.

Por ello, se ha introducido una regla de FW en Google Cloud donde a los puertos TCP se les ha asignado el rango de puertos mencionado anteriormente. (En caso de introducir un puerto fuera de dicho rango sale del script y no se ejecuta nuestra aplicación monolítica, habría que volver a ejecutar el script introduciendo un puerto correcto).

-Llama a "productpage_monolith.py" (script de la app) con el puerto introducido (correctamente) en el paso anterior.

A continuación, introducimos en el navegador la ip pública de la instancia (MV) con el puerto introducido: http://(ip-publica):(puerto)/productpage obteniendo el resultado esperado.



Como podemos observar, el título de la aplicación es en nuestro caso 35 y la conexión se establece correctamente. Dicha aplicación está compuesta por dos servicios: uno para la página de productos y otro para la descripción de los productos.

-----Part2-----DESPLIEGUE DE UNA APLICACIÓN MONOLÍTICA USANDO DOCKER-----

Ahora se quiere desplegar la misma aplicación monolítica pero usando docker. Para ello, hemos creado el script "script2.py" entregado en el zip, dentro de la carpeta "part2", que automatiza la creación de las imágenes y contenedores docker. Además de, lógicamente, el fichero "Dockerfile", un script que se ejecuta al crear la imagen docker (se hace run desde el Dockerfile) llamado "docker.py" y un script "delete.py" que eliminará los contenedores e imágenes creadas de docker, (una vez que ha sido lanzada la aplicación con "script2.py").

Para probarlo, hemos creado una MV en Google Cloud y abierto su consola SSH en un nuevo navegador. Además, hemos instalado git con el comando "sudo apt-get install git" para poder clonar la carpeta con los scripts. A continuación, nos cambiamos al directorio de la carpeta de esta parte (scriptsCDPS/part2). (Vale hacerlo en la MV creada en la parte1 cambiando simplemente de directorio).

Ejecutamos el script con el comando "python3 script2.py". Este script:

-Construye la imagen de docker "35/product-page".

-Arranca el contenedor de docker "35-productpage".

La imagen (y por consiguiente el contenedor) se construye según el contenido del "Dockerfile". Este fichero:

-Contiene la variable de entorno "GROUP_NUMBER" que es nuestro número de grupo (35).

-Copia y ejecuta el script "docker.py" para la imagen durante su proceso de construcción.

-Expone el puerto 9080 a través del cual se va a acceder al servicio.

-Ejecuta con python3 "productpage_monolith.py" (script de la app) con el puerto 9080 tras haber inicializado el contenedor.

El script "docker.py" mencionado, que se utiliza para crear la imagen del contenedor. Este script:

-Clona la carpeta practica_creativa2 del github de la asignatura (https://github.com/CDPS-ETSIT/practica_creativa2.git).

-Instala pip y se instalan las dependencias de requirements con pip3.

-Extrae la variable de entorno "GROUP_NUMBER" creado en el Dockerfile.

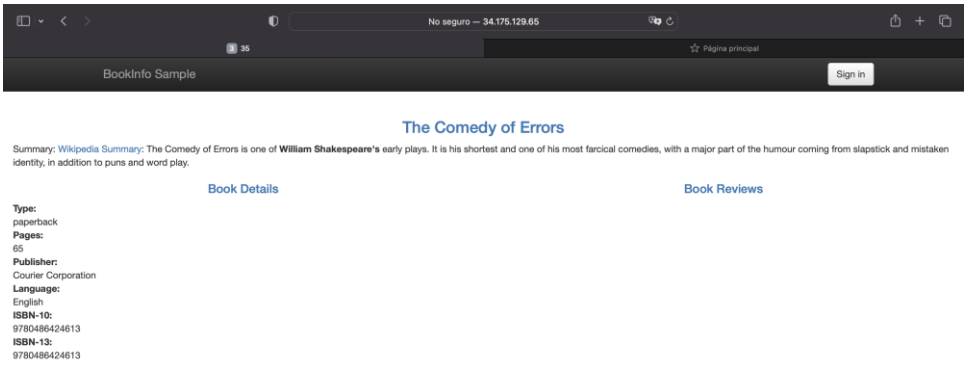
-Modifica el “productpage.html” para cambiar el título de la app por la variable de entorno.

El script "delete.py":

-Borra el contenedor creado "35-productpage".

-Borra la imagen creada "35/product-page".

A continuación de haber ejecutado el "script2.py", introducimos en el navegador la ip pública de la instancia (MV) con el puerto introducido: http://(ip-publica):9080/productpage obteniendo el resultado esperado.



Como podemos observar, el título de la aplicación es en nuestro caso 35 y la conexión se establece correctamente. Dicha aplicación está compuesta por dos servicios: uno para la página de productos y otro para la descripción de los productos.

>> Incluir la línea de comando del despliegue del contenedor en la memoria: nosotros incluimos los comandos que se piden en el "script2.py"

La imagen de docker se construye con: "sudo docker build -t 35/product-page ."

El contenedor de docker se arranca con: "sudo docker run --name 35-productpage -p9080:9080 35/product-page"

-----Part3-----SEGMENTACIÓN DE UNA APLICACIÓN MONOLÍTICA EN MICROSERVICIOS UTILIZANDO DOCKER-COMPOSE-----

Ahora se va a segmentar la aplicación, es decir, se va a separar cada servicio para que funcione de forma independiente usando docker-compose. Además de los servicios que teníamos anteriormente, productpage y details, se van a añadir dos más: reviews y ratings. Para ello, hemos creado el script "script3.py" entregado en el zip, dentro de la carpeta "part3", que automatiza la creación y lanzamiento de docker-compose. Además de,

lógicamente, el fichero "docker-compose.yaml" y los ficheros "Dockerfile" de cada servicio (contenidos respectivamente en las carpetas con sus nombres de servicio, excepto reviews, que se cogerá directamente de la "practica_creativa2"), un script dentro de la carpeta "productPage" que se ejecuta al crear la imagen docker para el servicio productpage (se hace run desde el Dockerfile de productpage) llamado "docker.py", exactamente igual que el utilizado en la parte 2, y un script "delete.py" que eliminará los contenedores e imágenes de docker creadas, (una vez que ha sido lanzada la aplicación con "script3.py").

Para probarlo, hemos creado una MV en Google Cloud y abierto su consola SSH en un nuevo navegador. Además, hemos instalado git con el comando "sudo apt-get install git" para poder clonar la carpeta con los scripts. A continuación, nos cambiamos al directorio de la carpeta de esta parte (scriptsCDPS/part3). (Vale hacerlo en la MV creada en la parte1 cambiando simplemente de directorio).

Ejecutamos el script con el comando "python3 script3.py". Este script:

-Clona la carpeta "practica_creativa2" del github de la asignatura (https://github.com/CDPS-ETSIT/practica_creativa2.git).

-Copia los ficheros "details.rb" (de la carpeta "details" de "practica_creativa2"), "ratings.js" y "package.json" (de "ratings") para almacenarlos en nuestras carpetas respectivas en "part3" junto al Dockerfile correspondiente.

-Compila y empaqueta los paquetes de la carpeta "reviews" (dentro de "practica_creativa2").

-Pide introducir la versión deseada para el servicio reviews: v1, v2 o v3

-Modifica el docker-compose.yaml para cambiar las variables de entorno de reviews según la versión introducida.

-Construye las imágenes con docker-compose y arranca los contenedores del docker-compose y lanza la aplicación con todos los servicios. (En caso de introducir una version inválida (que no sea v1, v2 o v3) sale del script y no se ejecuta nuestra aplicación, habría que volver a ejecutar el script introduciendo una versión válida).

La imagen del servicio productpage se construye con su Dockerfile que hace uso del script "docker.py", ambos ficheros están en la carpeta "productPage". Ambos son iguales que en la parte2, salvo que ya no se ejecuta "productpage_monolith.py", sino "productpage.py". (Por ello no nos detendremos en su contenido).

La imagen del servicio details se construye con su Dockerfile:

-Copia el fichero "details.rb" en la ruta "/opt/microservices/" dentro del contenedor.

-Tiene dos variables de entorno: "SERVICE_VERSION" con valor v1 y "ENABLE_EXTERNAL_BOOK_SERVICE" con valor true.

-Expone el puerto 9080 a través del cual se va a acceder al servicio.

-Ejecuta con ruby "details.rb" (código del servicio details) con el puerto 9080 tras haber inicializado el contenedor.

La imagen del servicio ratings se construye con su Dockerfile:

-Copia los ficheros "package.json" y "ratings.js" en la ruta "/opt/microservices/" dentro del contenedor.

-Tiene la variable de entorno: "SERVICE_VERSION" con valor v1.

-Instala las dependencias del json con npm install.

-Expone el puerto 9080 a través del cual se va a acceder al servicio.

-Ejecuta con node "ratings.js" (código del servicio ratings) con el puerto 9080 tras haber inicializado el contenedor.

La imagen del servicio reviews se construye con su Dockerfile dado en la ruta "practica_creativa2/bookinfo/src/reviews/reviews-wlpcfg". (Por ello no nos detendremos en su contenido).

El fichero "docker-compose.yaml" establece para cada servicio:

-La imagen que se va a utilizar para crear el contenedor.

-El nombre del contenedor que se va a crear.

-Las variables de entorno definidas en su Dockerfile con sus valores correspondientes.

-En el caso del servicio de productpage, los puertos 9080:9080; para el resto de servicios, la dependencia (details y reviews son hijos de productpage y ratings es hijo de reviews).

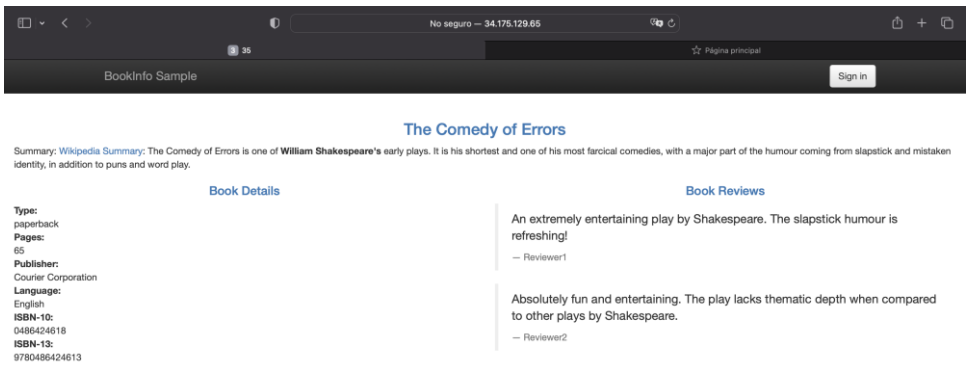
El script "delete.py":

-Borra los contenedores creados a partir de docker-compose.

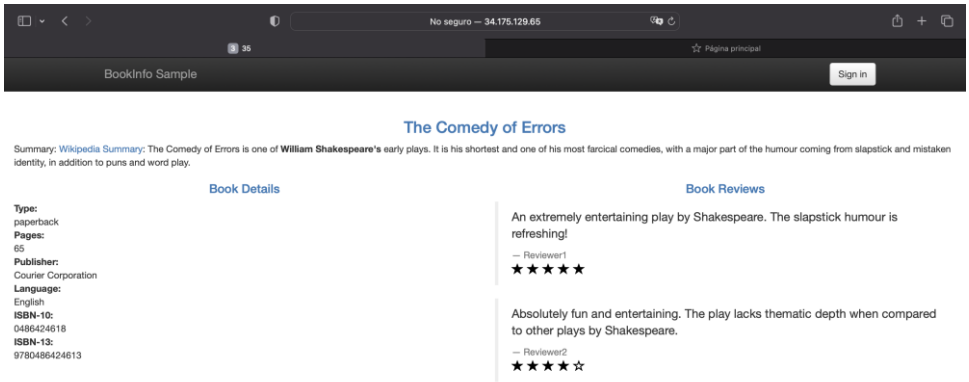
-Borra las imágenes creadas ("35/reviews", "35/ratings", "35/details", "35/productpage").

A continuación de haber ejecutado el "script3.py", introducimos en el navegador la ip pública de la instancia (MV) con el puerto 9080: http://(ip-publica):9080/productpage obteniendo el resultado esperado.

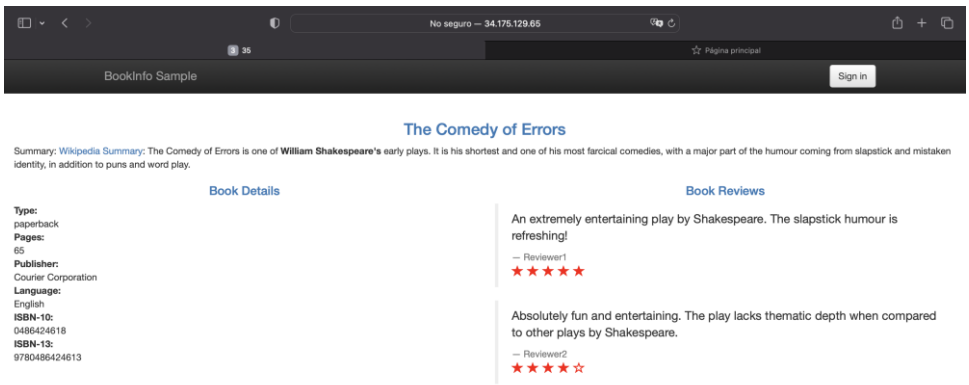
Seleccionando v1:



Seleccionando v2:



Seleccionando v3:



Como podemos observar, el título de la aplicación es en nuestro caso 35 y la conexión se establece correctamente. Dicha aplicación está compuesta por cuatro servicios: uno para la página de productos, uno para la descripción de los productos, uno para las críticas y otro para las valoraciones.

>> Incluir las diferencias con la versión de un único contenedor.

En una versión con varios contenedores, cada contenedor se encarga de un servicio específico. Esto facilita la administración, el mantenimiento y la escalabilidad de la aplicación, ya que se puede escalar cada contenedor de forma independiente según sea necesario.

En cambio, en una versión con un único contenedor, todas las responsabilidades están en un solo contenedor, lo que puede dificultar la administración y la escalabilidad, pero puede ser más sencillo de implementar y administrar debido a su simplicidad.

>> Incluir la línea del despliegue del docker-compose: nosotros incluimos los comandos que se piden en el "script3.py"

Las imágenes las construimos con "sudo docker-compose build"

Se despliegan los contenedores y se lanza la aplicación basada en microservicios con "sudo docker-compose up"

-----Part4-----DESPLIEGUE DE UNA APLICACIÓN BASADA EN MICROSERVICIOS UTILIZANDO KUBERNETES---

Ahora se pide desplegar la misma aplicación basada en microservicios pero utilizando Kubernetes, no docker-compose. Para ello, hemos creado el script "script4.py" entregado en el zip, dentro de la carpeta "part4", que automatiza la creación y lanzamiento de los servicios con Kubernetes. Además de, lógicamente, los ficheros yaml que definen los servicios de la aplicación "productpage.yaml" (dentro de la carpeta "productPage"), "details.yaml" (dentro de la carpeta "details"), "ratings.yaml" (dentro de la carpeta "ratings") y para el caso del servicio reviews tenemos la definición del servicio "reviews-svc.yaml", y los despliegues para las distintas versiones "reviews-v1-deployment.yaml", "reviews-v2-deployment.yaml" y "reviews-v3-deployment.yaml" (todos ellos dentro de la carpeta "reviews"); y un script "delete.py" que eliminará los pods (y servicios) de Kubernetes, (una vez que ha sido lanzada la aplicación con "script4.py").

Creamos las imágenes de productpage, details, ratings y las de cada versión de reviews y las subimos a nuestro repositorio de Docker-hub: sandracascantem/35. Las imágenes las tuvimos que nombrar como "sandracascantem/35:(servicio)" por ejemplo para productpage: sandracascantem/35:productpage, para la versión 1 de reviews: sandracascantem/35:reviews-v1, (indicado en los ficheros .yaml correspondientes). (No fue posible en Docker-hub respetar el nombre de las imágenes 35/servicio).

Para probarlo, hemos creado un clúster de Kubernetes en Google Cloud (GKE), de modo que se creen 5 nodos. Nos conectamos al clúster a través de la Google Shell, clonamos la carpeta con los scripts (https://github.com/sandracascantem/scriptsPC2). A continuación, nos cambiamos al directorio de la carpeta de esta parte (scriptsCDPS/part4).

Ejecutamos el script con el comando "python3 script4.py". Este script:

-Pide introducir la versión deseada para el servicio reviews: v1, v2 o v3.

-Construye los servicios de Kubernetes. Para productpage: "kubectl apply -f productPage/productpage.yaml", para details: "kubectl apply -f details/details.yaml", para ratings: "kubectl apply -f ratings/ratings.yaml", para reviews: "kubectl apply -f reviews/reviews-svc.yaml" y según la versión introducida (X) construye reviews/reviews-vX-deployment.yaml. (En caso de introducir una version inválida (que no sea v1, v2 o v3) sale del script y no se ejecuta nuestra aplicación, habría que volver a ejecutar el script introduciendo una versión válida).

Los ficheros "ratings.yaml" y los de reviews, "reviews-svc.yaml" y "reviews-vX-deployment.yaml" (X = 1, 2 o 3 para cada veersión), vienen dados como ejemplo en "practica_creativa2/bookinfo/platform", solo cambiamos el nombre de la imagen correspondiente subida al Docker-hub (como se explicó anteriormente). (Por ello no nos detendremos en su contenido).

El fichero "details.yaml" es como el anterior "ratings.yaml" pero cambiando los nombres por "details".

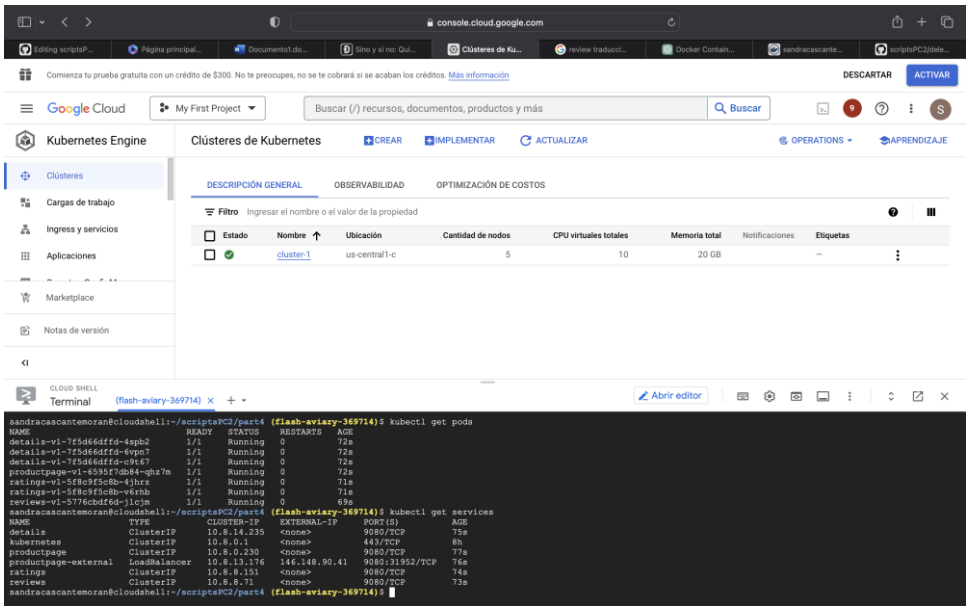
El fichero "productpage.yaml" añade un servicio (productpage-external) de tipo LoadBalancer para poder acceder a la aplicación por medio de una IP externa.

El script "delete.py":

-Borra los pods y servicios de Kubernetes creados, por ejemplo: "kubectl delete -f details/details.yaml", así para todos los posibles (sin diferenciar la versión que está corriendo de reviews).

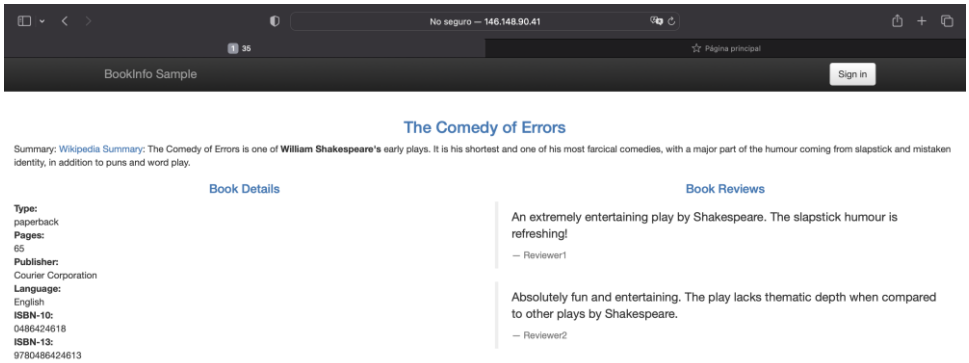
A continuación de haber ejecutado el "script4.py" en la Cloud Shell, comprobamos con "kubectl get pods" que los pods se hayan creado correctamente (ready: 1/1) y que estén corriendo (running); y con "kubectl get

services" obtenemos la IP externa de productpage-external. Como se puede ver en la siguiente imagen, en este caso la IP externa es 146.148.90.41

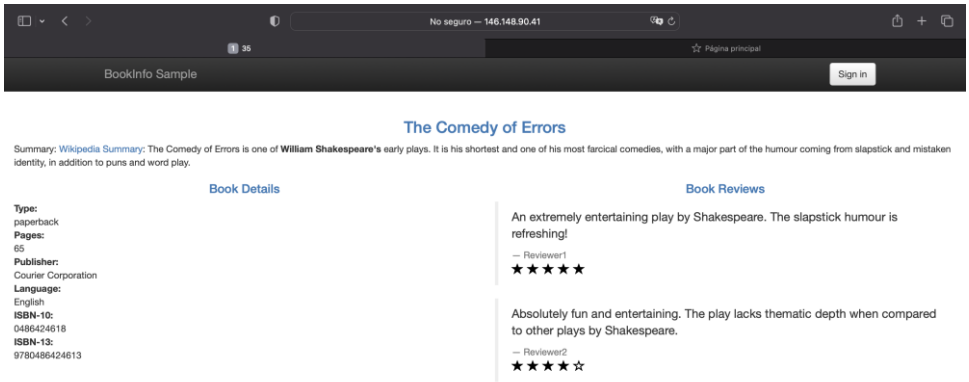


Acto seguido, introducimos en el navegador la ip externa anterior con el puerto 9080: http://(ip-externa):9080/productpage obteniendo el resultado esperado.

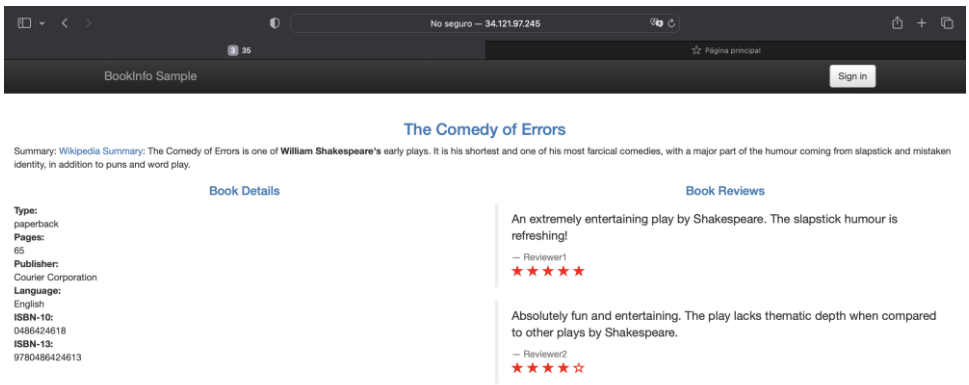
Seleccionando v1:



Seleccionando v2:



Seleccionando v3:



Como podemos observar, el título de la aplicación es en nuestro caso 35 y la conexión se establece correctamente. Dicha aplicación está compuesta por los servicios anteriores: uno para la página de productos, uno para la descripción de los productos, uno para las críticas y otro para las valoraciones, y se añade además un servicio de tipo LoadBalancer para acceder con una IP externa a la aplicación a través del microservicio productpage.

>> Incluir las diferencias que encuentra al crear los pods, así mismo la diferencia que ve para escalar esta última solución.

En Kubernetes, los pods son la unidad básica de escalabilidad y se encargan de ejecutar uno o más servicios, (en nuestro caso un pod para cada servicio). Los pods pueden ser creados, eliminados y reemplazados dinámicamente para garantizar la disponibilidad de la aplicación.

Kubernetes permite escalar las aplicaciones basadas en microservicios de manera fácil y eficiente. Se puede definir una cantidad deseada de replicas de un pod, y Kubernetes se encargará de crear o eliminar pods según sea necesario para cumplir con la cantidad deseada. Además, Kubernetes también permite la escalabilidad

automática, donde se puede definir una regla para ajustar dinámicamente la cantidad de réplicas en función de la utilización de recursos.

>> Incluir la línea del despligue de los ficheros de configuración y definición de pods y servicios en la infraestructura de kubernetes: nosotros incluimos los comandos que se piden en el "script4.py"

Para desplegar los ficheros de configuración para definir los pods y servicios utilizamos el comando "kubectl apply -f archivo.yaml. Siendo el archivo.yaml, para el servicio de productpage "productPage/productpage.yaml" (al estar en la carpeta "productPage"), para el servicio details "details/details.yaml" (al estar en la carpeta "details"), para el servicio de ratings "ratings/ratings.yaml" (al estar en la carpeta "ratings") y para el servicio de reviews necesitamos "reviews/reviews-svc.yaml" y la versión que se quiera, por ejemplo v2: "reviews/reviews-v2-deployment.yaml", (están en la carpeta "reviews").

Para comprobar que los pods se han generado correctamente se usa el comando "kubectl get pods" (comprobar ready: 1/1 y que están en estado running).

Para comprobar los servicios que están desplegados usamos el comando "kubectl get services" (de aquí es donde obteníamos la IP externa para poder acceder a la aplicación).