

UNIVERSIDAD DE GUADALAJARA
Centro Universitario de Ciencias Exactas e Ingenierías

Computación Tolerante a Fallas



Ejemplo con Kubernetes

Nombre: Castillo Mares Gilberto

Código: 213330514

Profesor: Dr. Michel Emanuel López Franco

2024-B

Índice

Contenido

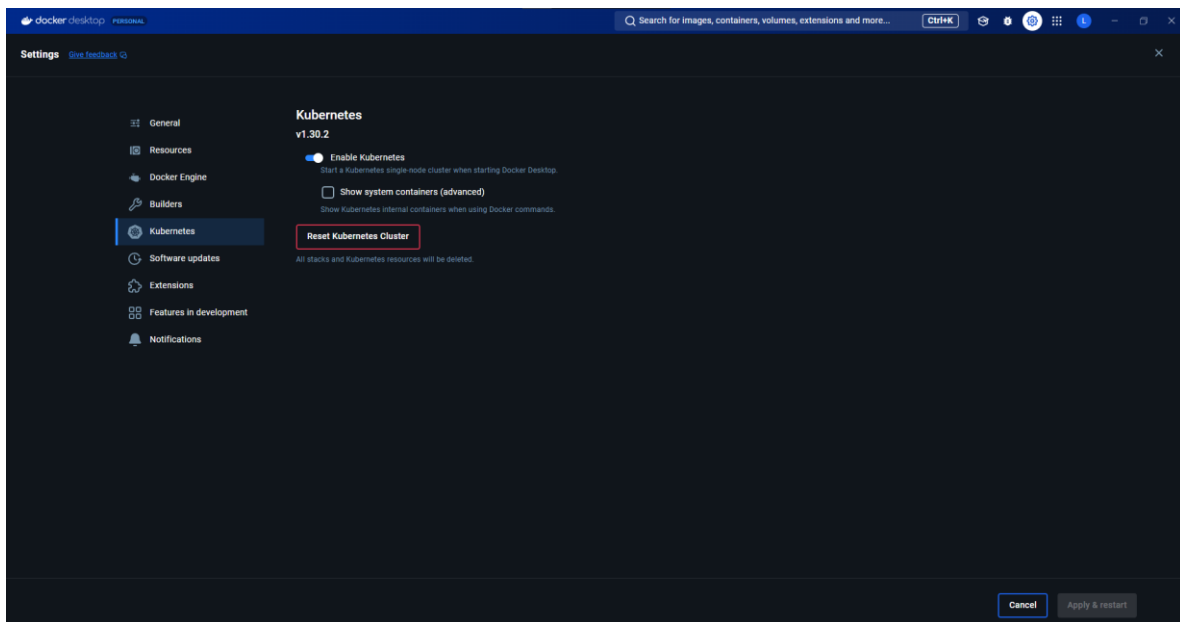
Contenido	3
-----------------	---

Contenido

Veremos entonces en este reporte, cómo hacer un ejemplo sencillo en el cual crearemos una API con Flask usando Docker y después importándola en un clúster de Kubernetes.

Estaré usando Python para crear la aplicación la cual mostrara solamente la cadena de texto “Hello World” y mostrare algunas modificaciones y como hacer los cambios para que surtan efecto. También usaré Visual Studio y Docker Desktop.

Primero una vez teniendo Docker Desktop, el programa cuenta con una opción para activar Kubernetes en esta misma para poder utilizar sus comandos.



En ajustes podemos encontrar la opción de activar Kubernetes y así un clúster, después aplicamos y reiniciamos.

Ahora instalamos la biblioteca de Flask en Python la cual nos permitira crear la aplicación.

Nota: Es recomendable utilizar un entorno virtual en Python

Una vez teniendo un entorno virtual podemos ejecutar el siguiente comando para instalar las dependencias de Flask:

```
pip install Flask
```

Ya instalado, para aplicar los cambios, reiniciamos Visual Studio cerrándolo y volviéndolo a abrir.

Entonces ahora si comenzaremos a crear la aplicación:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/v1/hello', methods=['GET'])
def hello():
    return jsonify(message="Hello, World!")

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

Esta API tiene un solo endpoint, /api/v1/hello, que devuelve un mensaje JSON de saludo.

Está colocando en el puerto 5000 y el host como 0.0.0.0 ya que significa que la aplicación está escuchando en todas las interfaces de red del contenedor, lo que permite que Kubernetes pueda redirigir tráfico desde el puerto del nodo a este puerto del contenedor.

A este script de Python lo llamamos **app.py**

Ahora debemos crear un **Dockerfile**, para empaquetar la API en una imagen de Docker. En Visual Studio solamente creamos un nuevo archivo y de nombre **Dockerfile** y listo. Dentro de este archivo agregamos los siguientes ajustes:

```
# Usa una imagen base de Python con la versión 3.9
FROM python:3.9

# Establece el directorio de trabajo en el contenedor
WORKDIR /app
```

```
# Copia el archivo de requerimientos (para instalar dependencias)
COPY requirements.txt /app

# Instala las dependencias usando pip
RUN pip install -r requirements.txt

# Copia el código fuente al contenedor
COPY app.py /app

# Expone el puerto 5000 (puerto en el que Flask ejecutará la API)
EXPOSE 5000

# Comando para correr la aplicación
CMD ["python", "app.py"]
```

Este Dockerfile realiza los siguientes pasos:

1. Usa una imagen base de Python.
2. Establece un directorio de trabajo (/app).
3. Copia e instala los paquetes necesarios.
4. Copia el archivo app.py.
5. Expone el puerto 5000 para que la API sea accesible.
6. Ejecuta la aplicación Flask.

Y ahora debemos crear un archivo de texto con el nombre ***requirements.txt*** para instalar Flask.

```
Flask==3.0.3
```

Y esto es lo único que contendrá.

Comprobaremos los archivos que hay en nuestra carpeta, todo debe estar en la misma. Podemos ejecutar el comando ***dir*** para ver que elementos contiene la carpeta actual.

```
(myvenv) PS F:\2024B\Ejemplo-Kubernetes> dir

Directorio: F:\2024B\Ejemplo-Kubernetes

Mode                LastWriteTime         Length Name
----                -
d-----          03/11/2024   02:01 p. m.          myvenv
-a----          03/11/2024   01:52 p. m.             66 .gitattributes
-a----          03/11/2024   02:03 p. m.            234 app.py
-a----          03/11/2024   02:04 p. m.            526 Dockerfile
-a----          03/11/2024   02:10 p. m.             14 requirements.txt
```

Debemos tener los 3 archivos: **app.py**, **Dockerfile** y **requirements.txt**

Construiremos ahora la imagen Docker y la probaremos, debemos ejecutar el siguiente comando:

```
docker build -t flask-api:latest .
```

Despues de algunos minutos veremos que se creó correctamente:

```
(myvenv) PS F:\2024B\Ejemplo-Kubernetes> docker build -t flask-api:latest .
[*] Building 268.1s (11/11) FINISHED
-> [internal] load build definition from Dockerfile
-> => transferring dockerfile: 565B
-> [internal] load metadata for docker.io/library/python:3.9
-> [auth] library/python:pull token for registry-1.docker.io
-> [internal] load .dockerignore
-> => transferring context: 2B
-> [1/5] FROM docker.io/library/python:3.9@sha256:ed8b9dd4e9f89c111f4bdb85a55f8c9f0e22796a298449380b15f627d9914095
-> => resolve docker.io/library/python:3.9@sha256:ed8b9dd4e9f89c111f4bdb85a55f8c9f0e22796a298449380b15f627d9914095
-> => sha256:63dc518f902b82e47f42908845205bcbdd2bea1a70e1f13f4fb0859fbfd91671 250B / 250B
-> => sha256:cccc6c1c4bf5a8539f053b01dda5a0fba46a5b04afdd30fb30dcacf526778824 19.84MB / 19.84MB
-> => sha256:4eb48115a0423399a647666a3212b3977f31d779480dca8d8d8f9bbfb35f92e4 6.16MB / 6.16MB
-> => sha256:ad1c7cfc347f5c86fc2678b58f6a8fb6c6003471405760532fc3240b9eb1b343 211.27MB / 211.27MB
-> => sha256:7aad5092c3b7a865666b14bef3d4d038282b19b124542f1a158c98ea8c1ed1b 64.39MB / 64.39MB
-> => sha256:da802df85c965baeca9d39869f9e2cbb3dc844d4627f413bfb2f2c3d6055988 24.05MB / 24.05MB
-> => sha256:7d98d813d54f6207a57721008a4081378343ad8f1b2db66c121406019171805b 49.56MB / 49.56MB
-> => extracting sha256:7d98d813d54f6207a57721008a4081378343ad8f1b2db66c121406019171805b
-> => extracting sha256:da802df85c965baeca9d39869f9e2cbb3dc844d4627f413bfb2f2c3d6055988
-> => extracting sha256:7aad5092c3b7a865666b14bef3d4d038282b19b124542f1a158c98ea8c1ed1b
-> => extracting sha256:ad1c7cfc347f5c86fc2678b58f6a8fb6c6003471405760532fc3240b9eb1b343
-> => extracting sha256:4eb48115a0423399a647666a3212b3977f31d779480dca8d8d8f9bbfb35f92e4
-> => extracting sha256:cccc6c1c4bf5a8539f053b01dda5a0fba46a5b04afdd30fb30dcacf526778824
-> => extracting sha256:63dc518f902b82e47f42908845205bcbdd2bea1a70e1f13f4fb0859fbfd91671
-> [internal] load build context
-> => transferring context: 328B
-> [2/5] WORKDIR /app
-> [3/5] COPY requirements.txt /app
-> [4/5] RUN pip install -r requirements.txt
-> [5/5] COPY app.py /app
-> exporting to image
-> exporting layers
-> => exporting manifest sha256:6f27af5b8e9bfbe35eb81210c11957321da65d4b6ca825aefae62a91b2d4cae4
-> => exporting attestation manifest sha256:510a945cad2095109a9fd4fb0d321ddb5a5b5b9d04908e2911c37811ea93f055
-> => exporting manifest list sha256:0909ef14cc29df9634bdc5b25581aa8992635b5521b749987ddb34747af74bea
-> => naming to docker.io/library/flask-api:latest
-> => unpacking to docker.io/library/flask-api:latest
WARNING: current commit information was not captured by the build: git was not found in the system: exec: "git.exe": executable file not found in %PATH%

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/5qsqh4lxjt9p0xwzyrhogw

What's next:
  View a summary of image vulnerabilities and recommendations -> docker scout quickview
```

Para este caso, tuve que crear una cuenta en el sitio web de Docker Hub la cual nos permite subir la imagen de Docker que vamos a crear. Ya que tengamos abierta la sesión en un navegador, en la terminal ejecutamos el comando:

```
docker login
```

Creamos el tag del repositorio el cual debe ser el mismo nombre cuando estemos por crear la imagen Docker:

```
docker tag flask-api:latest <tu-usuario-de-docker-hub>/flask-api:latest
```

Y ahora subimos la imagen a Docker Hub con el siguiente comando:

```
docker push <tu-usuario-de-docker-hub>/flask-api:latest
```

Lo siguiente es crear los archivos de configuración de Kubernetes, los cuales deben estar en la misma carpeta donde esta el código API, a estos archivos los llamaremos ***api-deployment.yaml*** y ***api-service.yaml*** estos archivos le dirán a Kubernetes cómo desplegar el API.

1. api-deployment.yaml

Este archivo define un Deployment, que controla el ciclo de vida de los contenedores de la API, así como el número de réplicas que deseas ejecutar.

Crea un archivo llamado api-deployment.yaml con el siguiente contenido:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-api-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask-api
  template:
```

```
metadata:
  labels:
    app: flask-api
spec:
  containers:
  - name: flask-api
    image: louisev/flask-api:v2
    ports:
    - containerPort: 5000
```

2. api-service.yaml

Este archivo define un Service, que expone el Deployment y permite el acceso desde fuera del clúster (utilizando el tipo LoadBalancer).

Crea un archivo llamado api-service.yaml con el siguiente contenido:

```
apiVersion: v1
kind: Service
metadata:
  name: flask-api-service
spec:
  selector:
    app: flask-api
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
  type: LoadBalancer
```

Nota: antes de continuar debemos saber el Port por el cual esta escuchando la imagen Docker, primero debemos ejecutar el siguiente comando para que la imagen comience a transmitir por el puerto que asignamos en la aplicación.

```
docker run -p 5000:5000 flask-api:latest
```


Ejecutamos el siguiente comando para saber el puerto:

```
kubectl get services
```

Lo cual nos dará los puertos:

```
(myvenv) PS F:\2024B\Ejemplo-Kubernetes> kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
flask-api-service	LoadBalancer	10.96.219.202	localhost	80:30537/TCP	38m
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	129m

Aquí nos interesa el puerto del servicio con el nombre que le pusimos a nuestra imagen Docker, en este caso es el puerto 80:30537.

Regresando al archivo api-service.yaml, editamos el contenido cambiando el tipo de cluster de LoadBalancer a NodePort y colocando el numero de puerto que nos arrojó el comando ingresado anteriormente:

```
apiVersion: v1
kind: Service
metadata:
  name: flask-api-service
spec:
  selector:
    app: flask-api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
      nodePort: 30537
  type: NodePort
```

En la terminal, asegúrate de estar en el mismo directorio que contiene api-deployment.yaml y api-service.yaml. Luego, ejecuta los siguientes comandos para aplicar los archivos de configuración en Kubernetes:

```
kubectl apply -f api-deployment.yaml
kubectl apply -f api-service.yaml
```

Estos comandos crearán tanto el Deployment como el Service en Kubernetes.

Para asegurarte de que el Deployment y el Service se hayan creado correctamente, puedes usar los siguientes comandos:

```
kubectl get deployments
kubectl get pods
```

Esto nos da la información de los archivos deployment y service y nos indica si estos están funcionando correctamente, si vemos los pods con el estatus **Running** entonces están funcionando como deben.

```
(myvenv) PS F:\2024B\Ejemplo-Kubernetes> kubectl get deployments
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
flask-api-deployment               2/2     2             2           25h
(myvenv) PS F:\2024B\Ejemplo-Kubernetes> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
flask-api-deployment-599d696959-f5rx2  1/1     Running   0           82m
flask-api-deployment-599d696959-wh9mr  1/1     Running   0           82m
```

Solo nos queda verificar si la API y los servicios están funcionando, podemos ejecutar el siguiente comando:

```
curl http://localhost:30537/api/v1/hello
```

O usando el navegador ingresamos el siguiente enlace:

```
http://localhost:30537/api/v1/hello
```

En la terminal podemos ver esto:

```
(myvenv) PS F:\2024B\Ejemplo-Kubernetes> curl http://localhost:30537/api/v1/hello

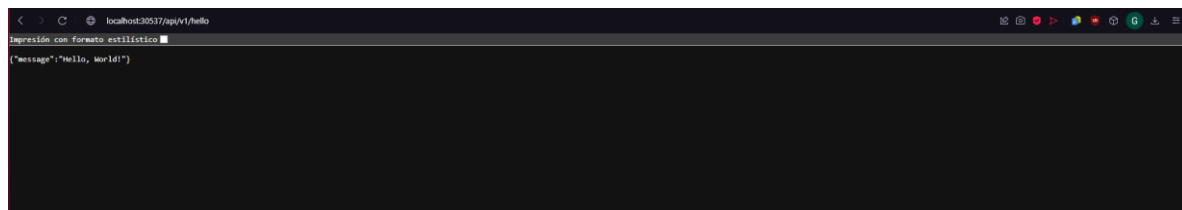
StatusCode      : 200
StatusDescription : OK
Content         : {"message":"Hello, World!"}

RawContent      : HTTP/1.1 200 OK
                  Connection: close
                  Content-Length: 28
                  Content-Type: application/json
                  Date: Sun, 03 Nov 2024 21:41:57 GMT
                  Server: Werkzeug/3.1.1 Python/3.9.20

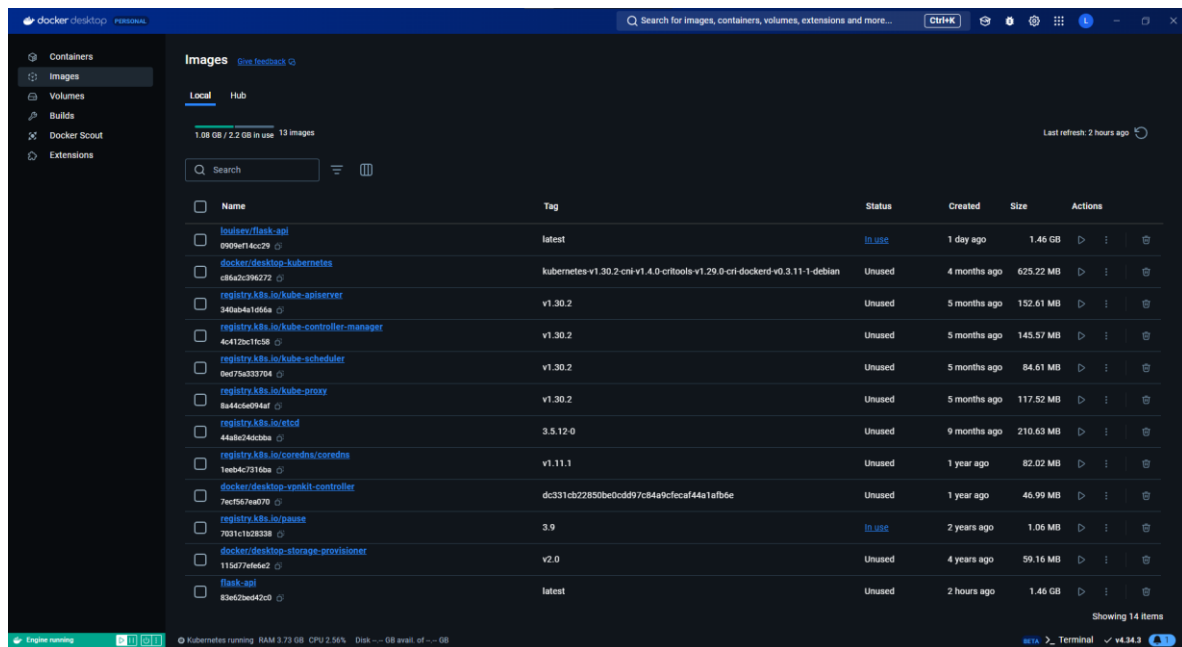
                  {"message":"Hello, World!"}

Forms           : {}
Headers         : {[Connection, close], [Content-Length, 28], [Content-Type, application/json], [Date, Sun, 03 Nov 2024 21:41:57 GMT]...}
Images          : {}
InputFields     : {}
Links           : {}
ParsedHtml      : mshhtml.HTMLDocumentClass
RawContentLength : 28
```

O en el navegador esto:



En Docker Desktop podemos detener los contenedores siempre que queramos así como monitorear su estado:



Y así podemos tener una API sencilla utilizando Docker y Kubernetes.

En el caso de que queramos modificar el código un poco, debemos hacerlo en el código fuente de la app, crear una etiqueta diferente para la imagen y volverla a subir. Para esto hacemos lo siguiente:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/v1/hello', methods=['GET'])
def hello():
    return jsonify(message="Hello, Worlddddddd!")

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

Aquí solamente modifiqué el mensaje que muestra el JSON. Guardamos el archivo.

Asigna una nueva etiqueta con un número de versión cada vez que actualices la imagen (por ejemplo, flask-api:v2 en lugar de latest). Esto asegura que Kubernetes descargue la nueva imagen.

```
docker build -t flask-api:v2 .
```

Luego, para Docker Hub:

```
docker tag flask-api:v2 <tu-usuario-de-docker-hub>/flask-api:v2
docker push <tu-usuario-de-docker-hub>/flask-api:v2
```

Actualizamos api-deployment.yaml para que apunte a esta nueva etiqueta:

```
containers:
  - name: flask-api
    image: louisev/flask-api:v2
    ports:
      - containerPort: 5000
```

Aplicamos el despliegue con el archivo actualizado:

```
kubectl apply -f api-deployment.yaml
```

Y ya deberíamos poder ver la modificación en el navegador o en la terminal:

```
(myvenv) PS F:\2024B\Ejemplo-Kubernetes> curl http://localhost:30537/api/v1/hello

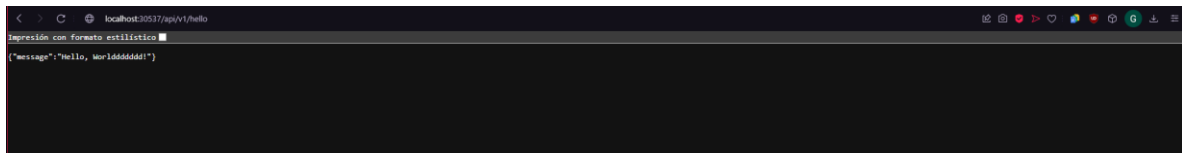
StatusCode      : 200
StatusDescription : OK
Content         : {"message":"Hello, Worlddddddd!"}

RawContent      : HTTP/1.1 200 OK
                  Connection: close
                  Content-Length: 34
                  Content-Type: application/json
                  Date: Mon, 04 Nov 2024 23:05:21 GMT
                  Server: Werkzeug/3.1.1 Python/3.9.20

                  {"message":"Hello, Worlddddddd!"}

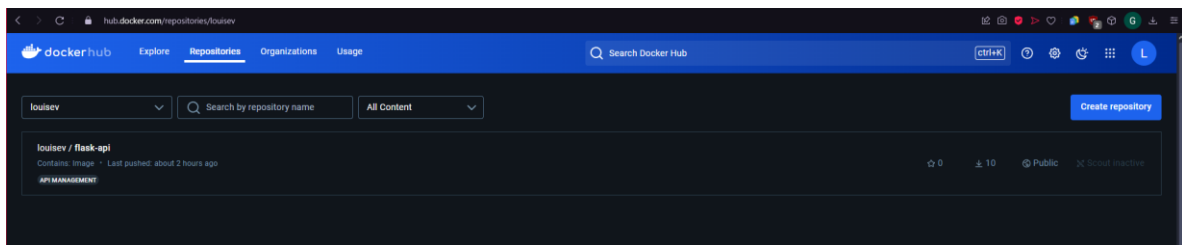
Forms           : {}
Headers         : {[Connection, close], [Content-Length, 34], [Content-Type, application/json], [Date, Mon, 04 Nov 2024 23:05:21 GMT]...}
Images          : {}
InputFields     : {}
Links           : {}
ParsedHtml      : mshtml.HTMLDocumentClass
RawContentLength : 34
```

O en el navegador:



A screenshot of a web browser window. The address bar shows 'localhost:30537/api/v1/hello'. The page content displays the JSON response: {"message":"Hello, Worlddddddd!"}.

En el sitio web de Docker Hub podemos ver el repositorio que hemos creado:



A screenshot of the Docker Hub website. The page shows the repository 'louisv / flask-api'. It indicates that the repository contains an image and was last pushed about 2 hours ago. The repository is public and has 0 stars and 10 downloads. The page also shows a 'Create repository' button and a search bar.