

Documentación

1. Creación y empaquetamiento de imagen en docker

Paso 1. Descarga Docker de su página oficial e instálalo.

Paso 2. Crear un archivo que contenga las dependencias necesarias para la ejecución de la aplicación. Guardar como Dockerfile en la carpeta de la aplicación.

```
#usar imagen de Node.js como base
FROM node:22-alpine

# directorio de trabajo del contenedor
WORKDIR /app

#copiar package.json y package-lock.json al contenedor
COPY package*.json ./

#instalar las dependencias
RUN npm install

#copiar el resto del código al contenedor
COPY . .









#puerto en el que se ejecutara la API
EXPOSE 3000

#comando para iniciar la app
CMD ["node","server.js"]
```

Paso 3. Una vez completada la configuración de la imagen se procede a construirla mediante el comando `Docker build -t nombre-de-la imagen.` en la terminal

```
docker build -t API-NODE .|
```

Paso 4. Construida la imagen procedemos a iniciarla el cual creara un contenedor

		api-node	latest	e169566e42f7		22 hours ago	244.08 MB	
		vibrant_brown		e7e82f9ab4e9		api-node	3000:3	

2. Despliegue en Kubernetes

Paso 1. Instalar minikube con el siguiente comando en modo administrador en la terminal.

```
PS C:\Users\USUARIO> choco install minikube -y
Chocolatey v2.3.0
3 validations performed. 2 success(es), 1 warning(s), and 0 error(s).
```

Paso 2. Descarga la versión adecuada de kubectl con el siguiente comando:

```
PS C:\Users\USUARIO> minikube kubectl -- get po -A
> kubectl.exe.sha256: 64 B / 64 B [-----] 100.00% ? p/s 0s
> kubectl.exe: 56.13 MiB / 56.13 MiB [-----] 100.00% 9.92 MiB p/s 5.9s
```

Paso 3. Crear un clúster con n número de nodos

```
PS C:\Users\USUARIO> minikube start --nodes=4
🐳 minikube v1.35.0 on Microsoft Windows 11 Pro 10.0.22631.4751 Build 22631.4751
🌟 Controlador de nodos configurado automáticamente
```

Nota: con el siguiente comando se confirma la creación de los nodos

```
PS C:\Users\USUARIO> kubectl get nodes
NAME                STATUS    ROLES    AGE   VERSION
minikube            Ready    control-plane   8m29s   v1.32.0
minikube-m02        Ready    <none>         7m58s   v1.32.0
minikube-m03        Ready    <none>         7m30s   v1.32.0
minikube-m04        Ready    <none>         7m4s    v1.32.0
```

Paso 4. Crea una cuenta en Docker hub para subir la imagen con el comando

Docker login

Y Etiqueta la imagen colocando el nombre de tu usuario seguido del repositorio con el comando

docker tag api-node:latest usuario/api-node:latest

Paso 5. Subir la imagen a Docker hub digitando el comando

docker push usuario/api-node:latest

Paso 6. Verificar que la imagen se subió correctamente

```
Name
-----
marce224/api-node
```

Paso 7. Crear el deployment con las 2 réplicas y distribuido en varios nodos, para ello crea un archivo Deployment.yml dentro del API en visual studio en una carpeta llamada Kubernetes

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: api-rest-deployment
5  spec:
6    replicas: 2
7    selector:
8      matchLabels:
9        role: api-rest
10   minReadySeconds: 20
11   strategy:
12     rollingUpdate:
13       maxUnavailable: 0
14       maxSurge: 1
15     type: RollingUpdate
16   template:
17     metadata:
18       labels:
19         role: api-rest
20     spec:
21       containers:
22         - name: competent-austin
23           image: marce224/api-node:latest
24           ports:
25             - containerPort: 3000
26

```

Paso 8. Desplegar el api en kubernetes aplicando el archivo creado deployment.yml con el siguiente comando

```

PS C:\Users\USUARIO\Desktop\LIS INVESTIGACION 1\API REST> kubectl apply -f kubernetes/Deployment.yml
>>
deployment.apps/api-rest-deployment unchanged

```

Nota: para confirmar los pods (instancias) creadas en este caso 2 y en que nodo se asignaron, digitar el siguiente código:

```

PS C:\Users\USUARIO\Desktop\LIS-Investigacion-1-main\LIS-Investigacion-1-main> kubectl get pods
>>

```

NAME	READY	STATUS	RESTARTS	AGE
api-rest-deployment-7dc7dc88c4-dzsk9	1/1	Running	0	69s
api-rest-deployment-7dc7dc88c4-g2mgm	1/1	Running	0	92s

3. Implementación de balanceo de carga

Paso 1. Crear un archivo en la carpeta kubernetes llamado Service.yml para exponer la API fuera del clúster e implementar el balanceo de carga

```

apiVersion: v1
kind: Service
metadata:
  name: api-service-loadbalncer
spec:
  selector:
    role: api-rest
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: LoadBalancer

```

Es importante que el spec/selector/role sea el mismo nombre que en el archivo deployment.yml ya que esto asegura que solo los pods con ese nombre sean considerados para recibir el tráfico.

El trafico externo llega al puerto 80 del Service y se reenvía al puerto 3000 en los Pods, donde está el API.

Paso 2. Desplegar el archivo creado con el siguiente comando.

```

PS C:\Users\USUARIO\Desktop\LI5 INVESTIGACION 1\API REST> kubectl apply -f .\kubernetes\Service.yml
service/api-service created

```

Nota: para verificar el estado del servicio y si kubernetes ha asignado una IP externa se digita el siguiente comando

```

PS C:\Users\USUARIO\Desktop\LI5-Investigacion-1-main\LI5-Investigacion-1-main> minikube service api-service-loadbalncer
--url
http://127.0.0.1:52279

```

4. Escalado Horizontal

Paso 1: Vamos instalar el **Metrics Server** el cual recopila datos de CPU y memoria de los pods y nodos, permitiendo que el **HPA** tome decisiones de escalado automático en Kubernetes.

```

PS E:\LI5-Investigacion-1-main> kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
serviceaccount/metrics-server unchanged
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader unchanged
clusterrole.rbac.authorization.k8s.io/system:metrics-server unchanged
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader unchanged
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator unchanged
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server unchanged
service/metrics-server unchanged
deployment.apps/metrics-server configured
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io unchanged
PS E:\LI5-Investigacion-1-main>

```

Paso 2: Vamos a crear un archivo YAML para el HPA. Este archivo definirá cómo Kubernetes debe escalar tu aplicación basándose en el uso de CPU o memoria.

```

1  apiVersion: autoscaling/v2beta2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: api-rest-hpa
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: api-rest-deployment
10   minReplicas: 2
11   maxReplicas: 10
12   metrics:
13   - type: Resource
14     resource:
15       name: cpu
16       target:
17         type: Utilization
18         averageUtilization: 50
19

```

Para este ejercicio, el HPA escalará el número de réplicas entre 2 y 10, basándose en el uso de CPU. Si el uso promedio de CPU supera el 50%, Kubernetes aumentará el número de réplicas.

Paso 3: Aplica el archivo YAML del HPA en tu clúster de Kubernetes usando kubectl:

```

PS E:\LIS-Investigacion-1-main\kubernetes> kubectl apply -f hpa.yml
horizontalpodautoscaler.autoscaling/api-rest-hpa created
PS E:\LIS-Investigacion-1-main\kubernetes> |

```

El **HPA** se creó correctamente y ahora escala automáticamente api-rest-deployment según el uso de CPU o memoria definido en hpa.yml.

¿Cómo el balanceador de carga gestiona el tráfico entre las réplicas y asegura la alta disponibilidad de la API, así como el sistema de autenticación garantiza el acceso seguro a los datos expuestos por la API?

El balanceador de carga en kubernetes usando el servicio de tipo LoadBalancer distribuye el tráfico de manera equitativa entre las réplicas del API, esto asegura la alta disponibilidad y distribución eficiente del tráfico de la siguiente manera:

1. El servicio en kubernetes crea un balanceador de carga que gestiona las solicitudes entrantes y las distribuye en los diferentes pods .

2. Mediante diferentes técnicas el balanceador de carga dirige las solicitudes a los pods disponibles de manera eficiente.
3. Si un pods falla, otros pods siguen manejando solicitudes sin interrupciones
4. Si se añaden más pods, el HPA ajusta dinámicamente el número de replicas en función de métricas como CPU o memoria.

La implementación del token JWT en el API garantiza la seguridad de los datos de la siguiente manera:

1. Cuando el usuario inicia sesión, se genera un token único, este token está firmado digitalmente para evitar manipulaciones.
2. El JWT incluye una fecha de caducidad, es decir que solo es válido por un tiempo limitado reduciendo riesgo de robo.
3. Cada vez que el cliente envía solicitud el token se envía y el servidor lo valida, si el token no es válido se rechaza.
4. El token se transmite a través de https para protegerlo de interceptación.