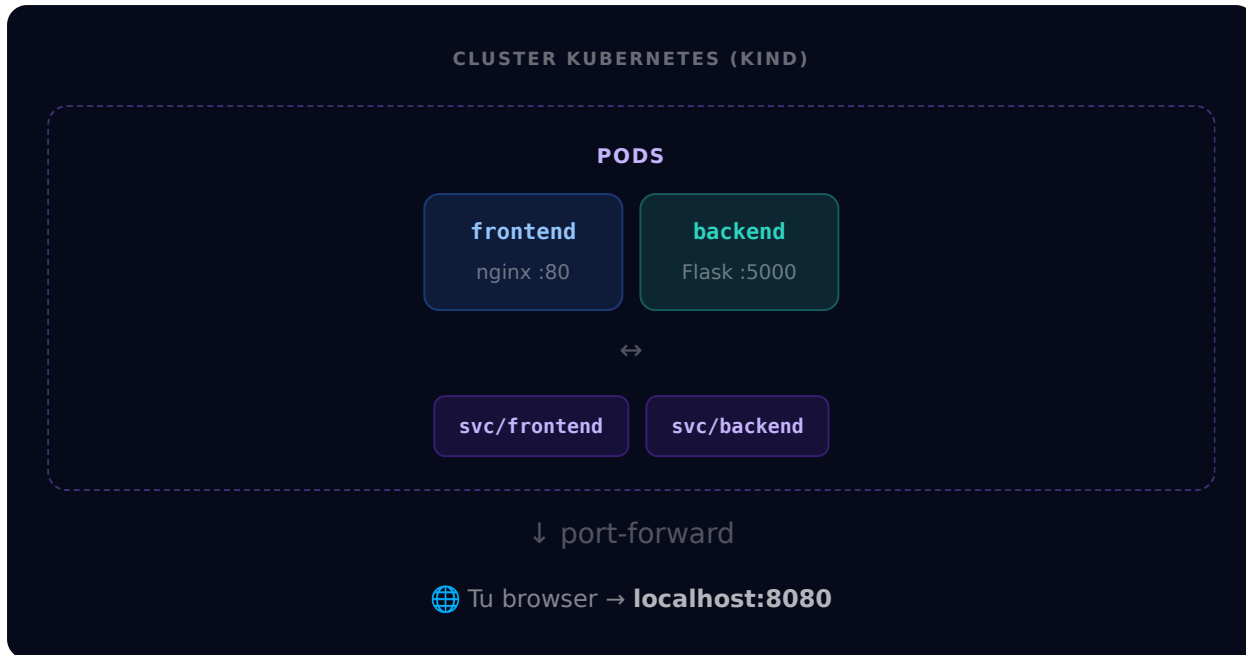


Lab: Frontend + Backend en Kubernetes

La misma app del lab Docker — pero ahora corriendo en un cluster Kubernetes local con kind.



ANTES DE EMPEZAR

- ☐ Docker Desktop corriendo
- ☐ Lab de Docker completado
- ☐ Terminal abierta

PARTE 1

Instalar kubectl y kind

1

Instalar kubectl y kind

Mac:

```
brew install kubectl kind
```

Windows (PowerShell como Admin):

```
choco install kubernetes-cli kind
```

Verificar:

```
$ kubectl version --client
$ kind version
```

PARTE 2

Descargar el proyecto y crear el cluster

2 Descargar la carpeta del lab

Todo está listo — los Dockerfiles, los YAML de Kubernetes y la app:

```
$ git clone https://github.com/micadevops/camino-devops-en-aws.git
$ cd camino-devops-en-aws/sesion3/sesion3-lab-home-kubernetes
```

```
sesion3-lab-home-kubernetes/
├─ backend/
│   └─ backend.py, requirements.txt, Dockerfile
├─ frontend/
│   └─ frontend.html, Dockerfile
├─ backend-deployment.yaml
├─ backend-service.yaml
├─ frontend-deployment.yaml
└─ frontend-service.yaml
```

3 Crear el cluster Kubernetes

```
$ kind create cluster --name lab-k8s

✓ Preparing nodes
✓ Starting control-plane
Set kubectl context to "kind-lab-k8s"
```

kind crea un cluster Kubernetes real dentro de un contenedor Docker. No necesitás servidores ni VMs.

4 Construir las imágenes y cargarlas al cluster

```
# Construir las imágenes Docker
$ docker build -t lab-backend:1.0 ./backend
$ docker build -t lab-frontend:1.0 ./frontend
```

```
# Cargarlas dentro del cluster kind
$ kind load docker-image lab-backend:1.0 --name lab-k8s
$ kind load docker-image lab-frontend:1.0 --name lab-k8s
```

En producción subirías las imágenes a ECR. Con kind, las cargamos directo al cluster local.

PARTE 3

Entender los archivos YAML

En Docker Compose tenías **1 archivo**. En Kubernetes tenés **4 archivos YAML** — un Deployment y un Service por cada servicio.

5 backend-deployment.yaml

Le dice a Kubernetes: "quiero 1 Pod corriendo con mi imagen de backend".

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 1                                # cuántas copias quiero
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        name: backend
        image: lab-backend:1.0              # la imagen que construimos
        ports:
          containerPort: 5000
        imagePullPolicy: Never              # usar la imagen local
```

6 backend-service.yaml

Le da una dirección fija al backend dentro del cluster.

```
apiVersion: v1
kind: Service
```

```

metadata:
  name: backend
spec:
  selector:
    app: backend    # conecta con los Pods que tengan app: backend
  ports:
    port: 5000
    targetPort: 5000

```

Los archivos de frontend (`frontend-deployment.yaml` y `frontend-service.yaml`) son iguales pero con imagen `lab-frontend:1.0` y puerto `80` .

PARTE 4

Desplegar y ver la app

7 Aplicar todos los YAML

```

$ kubectl apply -f .

deployment.apps/backend created
service/backend created
deployment.apps/frontend created
service/frontend created

```

`kubectl apply -f .` = aplicá todos los YAML de esta carpeta. Es el equivalente de `docker compose up` .

8 Verificar que está corriendo

```

$ kubectl get pods

```

NAME	READY	STATUS	AGE
backend-7d4f8b6c5-x2k9p	1/1	Running	15s
frontend-5c8d9f7b2-m3j7q	1/1	Running	15s

Si ves **STATUS: Running** en ambos, todo está bien.

9 Acceder desde el browser

Abrí **dos terminales** y corré un port-forward en cada una:

```

# Terminal 1 – backend

```

```
$ kubectl port-forward svc/backend 5000:5000
```

```
# Terminal 2 – frontend
```

```
$ kubectl port-forward svc/frontend 8080:80
```

DEJÁ AMBAS TERMINALES ABIERTAS

Mientras estén corriendo, los puertos están disponibles. Si cerrás la terminal, se corta el acceso.

Ahora abrí en tu browser: <http://localhost:8080>



La tarjeta trofeo aparece!

La misma app de siempre, pero corriendo en un cluster Kubernetes local.

BONUS

Los superpoderes de Kubernetes

B1 Escalar a 3 réplicas

```
$ kubectl scale deployment backend --replicas=3
```

```
$ kubectl get pods
```

backend-7d4f8b6c5-x2k9p	1/1	Running	5m
backend-7d4f8b6c5-a1b2c	1/1	Running	5s
backend-7d4f8b6c5-d3e4f	1/1	Running	5s

3 copias del backend corriendo. El Service reparte el tráfico entre las 3.

B2 Ver qué Pod te responde

El backend devuelve el nombre del Pod en el campo `"pod"`. Probá llamar varias veces:

```
$ curl http://localhost:5000/info?nombre=Mica
```

```
{"nombre": "Mica", "pod": "backend-7d4f8b6c5-x2k9p"}
```

```
$ curl http://localhost:5000/info?nombre=Mica
```

```
{"nombre": "Mica", "pod": "backend-7d4f8b6c5-a1b2c"}
```

```
$ curl http://localhost:5000/info?nombre=Mica
```

```
{"nombre": "Mica", "pod": "backend-7d4f8b6c5-d3e4f"}
```

Cada vez responde un Pod distinto. El Service reparte el tráfico automáticamente entre las 3 réplicas.

B3 Ver los logs de un Pod

```
# Logs de un Pod específico
$ kubectl logs backend-7d4f8b6c5-x2k9p

# Logs en tiempo real (follow)
$ kubectl logs -f backend-7d4f8b6c5-x2k9p

# Logs de TODOS los Pods del backend a la vez
$ kubectl logs -l app=backend
```

`-l app=backend` usa la label para filtrar — muestra los logs de los 3 Pods juntos.

B4 Self-healing: matá un Pod

```
# Elejí un Pod y eliminalo
$ kubectl delete pod backend-7d4f8b6c5-x2k9p
pod "backend-7d4f8b6c5-x2k9p" deleted

# Mirá los Pods inmediatamente
$ kubectl get pods
backend-7d4f8b6c5-a1b2c    1/1    Running    2m
backend-7d4f8b6c5-d3e4f    1/1    Running    2m
backend-7d4f8b6c5-g5h6i    1/1    Running    3s ← nuevo!
```

Dijiste "quiero 3". Mataste uno — K8s creó otro en **segundos**. Eso es self-healing.

B5 Limpiar todo

```
$ kubectl delete -f .
$ kind delete cluster --name lab-k8s
```



Lab completado

Cluster creado, app desplegada con YAML, viste qué Pod te responde, logs, escalado y self-healing.