Computación de Altas Prestaciones (2025-2026)

Realizado por: Javier Agüero y Marouane Chahbar

Fecha: 23/11/2025

1 Práctica 3: nuevas tendencias en HPC, Big Data. PaaS y Kubernetes

Ejercicio 1: Tutorial PHP

Lea la información al respecto del cluster que hay en Moodle. Realice el tutorial Example: Deploying PHP Guestbook application with Redis. Para su realización, descargue los ficheros YAML que se muestran, **adáptelos** al entorno y explique los cambios que ha realizado.

-Para empezar, hemos descargado el archivo *redis-leader-deployment.yaml,* que es un deployment encargado de mantener activa la instancia principal de la base de datos Redis (el programa en sí, la base de datos). Es necesario para garantizar la disponibilidad del servicio de escritura de la aplicación. Con este archivo .yaml le pedimos a Kubernetes que se asegure de que se cumplen siempre las condiciones definidas por éste, de forma que se encarga de gestionarlo por nosotros (y así con los demás archivos).

El fichero original que nos daban no incluía la especificación de recursos límite. Dado que el cluster de prácticas impone que debe haber un *LimitRange* por namespace, tuvimos que añadir a la sección *resources* tanto la sección *request* como *limits* (recursos que se dan y límites de recursos respectivamente). Sin esta modificación el planificador de Kubernetes rechazaba la creación del Pod por no cumplir las políticas de admisión del clúster.

Nuestro archivo queda ahora de esta forma:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: redis-leader
 labels:
   app: redis
   role: leader
   tier: backend
spec:
 replicas: 1
 selector:
   matchLabels:
     app: redis
 template:
   metadata:
     labels:
       app: redis
       role: leader
       tier: backend
   spec:
     containers:
     - name: leader
```

```
        image: docker.io/redis:6.0.5
        resources:
          # EL MÍNIMO
          requests:
            cpu: 100m
            memory: 100Mi
          # EL MÁXIMO, si se supera, el contenedor puede ser terminado
          limits:
            cpu: 200m
            memory: 200Mi
        ports:
        - containerPort: 6379
```

Hemos dado unos límites del doble de recursos pedidos, los cuales están muy lejos de los límites que nos dan del cluster y es seguro.

-Ahora vamos a crear el servicio para el "líder" de Redis. Lo haremos usando el *redis-leader-service.yaml.* Que nos definirá como poder acceder a los Pods, ya que de normal no podríamos acceder a ellos a través de sus IPs propias ya que son efímeras. De esta forma, el archivo queda así:

```
apiVersion: v1
kind: Service
metadata:
 name: redis-leader
 labels:
   app: redis
   role: leader
   tier: backend
spec:
 ports:
 - port: 6379
   targetPort: 6379
 selector:
   app: redis
   role: leader
   tier: backend
```

Al no aparecer la línea *Type* se asume por defecto que es *ClusterIP,* es decir, que este servicio sólo tendrá una IP interna, accesible únicamente por otros Pods dentro del clúster. De esta forma nadie puede entrar directamente a la base de datos desde fuera. Por lo que este archivo garantiza que aunque el Pod líder muera y sea reemplazado por otro con distinta IP, el resto de la aplicación siga sabiendo dónde enviar nuevos datos.

-Ahora seguimos con el *redis-follower-service.yaml* de la misma forma que el anterior, este servicio sólo tendrá una dirección IP interna, y los límites de los recursos son correctos, asi que no necesitamos cambiar nada del archivo que nos dan por defecto. Su función es balancear la carga entre los followers para que la web responda rápido:

```
apiVersion: v1
kind: Service
metadata:
 name: redis-follower
 labels:
   app: redis
   role: follower
   tier: backend
spec:
 ports:
   # the port that this service should serve on
 - port: 6379
 selector:
   app: redis
   role: follower
   tier: backend
```

-Ahora continuamos con el *frontend-deployment*.yaml. Al igual que lo anterior, el *frontend* se despliega usando un despliegue de Kubernetes. Y es como si fuera el cerebro que decide con quién hablar. Se comunicará con el *redis follower* o *leader services* dependiendo de que la solicitud sea de lectura o escritura:

```

apiVersion: apps/v1
kind: Deployment
metadata:
 name: frontend
spec:
 replicas: 3
 selector:
   matchLabels:
       app: guestbook
       tier: frontend
 template:
   metadata:
     labels:
       app: guestbook
```

```
      tier: frontend
  spec:
    containers:
    - name: php-redis
      image:
us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:v5
      env:
      - name: GET_HOSTS_FROM
        value: "dns"
      resources:
        # EL MÍNIMO
        requests:
          cpu: 100m
          memory: 100Mi
        # EL MÁXIMO, si se supera, el contenedor puede ser terminado
        limits:
          cpu: 200m
          memory: 200Mi
      ports:
      - containerPort: 80
```

Aquí solo hemos tenido que añadir el límite de uso de recursos.

-Ahora vamos a crear el *frontend-service.yaml*. El cual, va a hacer que la aplicación sea accesible externamente. Ya que el *service* ofrecerá una IP estable y externa gracias a LoadBalancer. Además recibirá el tráfico de los usuarios en el puerto 80 y lo repartirá equitativamente entre las 3 réplicas que creó el Deployment:

```
apiVersion: v1
kind: Service
metadata:
 name: frontend
 labels:
   app: guestbook
   tier: frontend
spec:
 # if your cluster supports it, uncomment the following to
automatically create
 # an external load-balanced IP for the frontend service.
 type: LoadBalancer
 ports:
   # the port that this service should serve on
```

```
- port: 80
  targetPort: 80
selector:
  app: guestbook
  tier: frontend
```

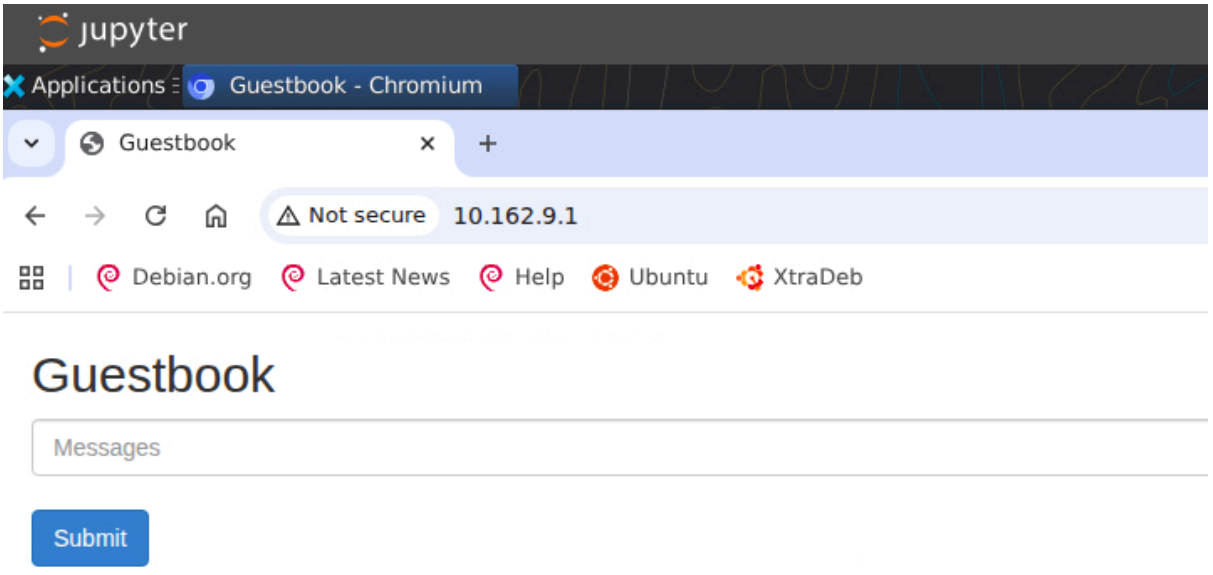
De esta forma, hemos añadido en el campo *spec* el type *loadbalancer*. Con esto nuestra web será accesible desde fuera ya que le pedimos a *MetalLB* (el balanceador de carga de nuestro entorno) que nos asigne una IP externa del rango reservado para estudiantes. Sin esto, el servicio sería *ClusterIP* por defecto (interno).

Con ello si ejecutamos *kubectl get pods* podemos ver lo siguiente:

```
NAME                              READY   STATUS    RESTARTS   AGE
frontend-9ff57fc69-2k56z          1/1     Running   0          7s
frontend-9ff57fc69-88snc          1/1     Running   0          7s
frontend-9ff57fc69-d5gsj          1/1     Running   0          7s
redis-follower-5bf955ffdc-mpnnk   1/1     Running   0          17s
redis-follower-5bf955ffdc-qjpwq   1/1     Running   0          17s
redis-leader-f8b565846-q8nnq      1/1     Running   0          18s
```

De la misma manera, al ejecutar *kubect get service frontend* obtenemos la dirección IP externa a la que podemos acceder en el navegador:

```
kun@kuns-Laptop:~/Dev/GitHub/CAP_LAB/practica3$ kubectl get service frontend
NAME       TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
frontend   LoadBalancer   10.43.21.80   10.162.9.1    80:31945/TCP   4m25s
```



Por último en el tutorial nos hablan de cómo escalar hacia arriba o abajo la web que hemos creado según necesitemos. De forma que si ejecutamos: *kubectl scale deployment frontend --replicas=5* escalamos el número de réplicas de frontend a 5:

```
NAME                              READY   STATUS    RESTARTS   AGE
frontend-9ff57fc69-2k56z          1/1     Running   0          13m
frontend-9ff57fc69-88snc          1/1     Running   0          13m
frontend-9ff57fc69-d5gsj          1/1     Running   0          13m
frontend-9ff57fc69-ttz9p          1/1     Running   0          55s
frontend-9ff57fc69-wl27f          1/1     Running   0          55s
redis-follower-5bf955ffdc-mpnnk   1/1     Running   0          13m
redis-follower-5bf955ffdc-qjpwq   1/1     Running   0          13m
redis-leader-f8b565846-q8nnq      1/1     Running   0          13m
```

Pudiéndose escalar de la misma forma hacia abajo: *kubectl scale deployment frontend --replicas=2*

```
NAME                              READY   STATUS    RESTARTS   AGE
frontend-9ff57fc69-2k56z          1/1     Running   0          14m
frontend-9ff57fc69-88snc          1/1     Running   0          14m
redis-follower-5bf955ffdc-mpnnk   1/1     Running   0          14m
redis-follower-5bf955ffdc-qjpwq   1/1     Running   0          14m
redis-leader-f8b565846-q8nnq      1/1     Running   0          14m
```

Ejercicio 2: Metadatos, estado y especificación

Extraiga en formato YAML los datos de un despliegue (por ejemplo, del tutorial). Compare los tres campos principales que tiene un objeto en Kubernetes y describa el propósito de cada uno.

Primero vamos a extraer la información técnica de nuestro frontend y la guardaremos en el archivo *ejercicio2.yaml*. Para eso ejecutamos el comando: *kubectl get deployment frontend -o yaml > ejercicio2.yaml*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 annotations:
   deployment.kubernetes.io/revision: "1"
   kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{
},"name":"frontend","namespace":"cap1462-09"},"spec":{"replicas":3,"sel
```

ector":{"matchLabels":{"app":"guestbook","tier":"frontend"}},"template"
:{"metadata":{"labels":{"app":"guestbook","tier":"frontend"}},"spec":{"
containers":[{"env":[{"name":"GET_HOSTS_FROM","value":"dns"}],"image":"
us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:v5","name":
"php-redis","ports":[{"containerPort":80}],"resources":{"limits":{"cpu"
:"200m","memory":"200Mi"},"requests":{"cpu":"100m","memory":"100Mi"}}}]
}}}}

```
 creationTimestamp: "2025-11-24T22:09:09Z"
 generation: 1
 name: frontend
 namespace: cap1462-09
 resourceVersion: "68941812"
 uid: e1f59dc1-d852-4334-bb43-752e0cf69088
spec:
 progressDeadlineSeconds: 600
 replicas: 3
 revisionHistoryLimit: 10
 selector:
   matchLabels:
     app: guestbook
     tier: frontend
 strategy:
   rollingUpdate:
     maxSurge: 25%
     maxUnavailable: 25%
   type: RollingUpdate
 template:
   metadata:
     creationTimestamp: null
     labels:
       app: guestbook
       tier: frontend
   spec:
     containers:
     - env:
       - name: GET_HOSTS_FROM
         value: dns
       image:
us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:v5
       imagePullPolicy: IfNotPresent
       name: php-redis
       ports:
       - containerPort: 80
```

```
        protocol: TCP
      resources:
        limits:
          cpu: 200m
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 100Mi
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
status:
 availableReplicas: 3
 conditions:
 - lastTransitionTime: "2025-11-24T22:09:10Z"
   lastUpdateTime: "2025-11-24T22:09:10Z"
   message: Deployment has minimum availability.
   reason: MinimumReplicasAvailable
   status: "True"
   type: Available
 - lastTransitionTime: "2025-11-24T22:09:09Z"
   lastUpdateTime: "2025-11-24T22:09:10Z"
   message: ReplicaSet "frontend-9ff57fc69" has successfully
progressed.
   reason: NewReplicaSetAvailable
   status: "True"
   type: Progressing
 observedGeneration: 1
 readyReplicas: 3
 replicas: 3
 updatedReplicas: 3
```

Aquí podemos identificar tres campos principale: *metadata, spec* y *status.*

- ***Metadata****:* Es como si fuera el “DNI” del objeto, es decir, contienen los datos que identifican al objeto de forma única dentro del clúster. En nuestro caso el objeto se

llama *frontend* y vive en el namespace *cap1462-09* (nuestro espacio reservado). El *uid* hace referencia a ese "DNI" único de este objeto en la base de datos del clúster.

- ***Spec**:* define el estado deseado. Kubernetes leerá esto y hará todo lo posible para que se cumplan dichas especificaciones.

- ***Status***: refleja el estado actual observado por el sistema. Esta sección es solo de lectura para el usuario (la escribe el controlador de Kubernetes).

Así que de forma resumida, al inspeccionar el objeto, observamos el bucle de reconciliación de Kubernetes en acción: la sección spec solicita 3 réplicas con unos límites de recursos específicos, y la sección status confirma que el clúster ha logrado provisionar esas 3 réplicas (readyReplicas: 3), cumpliendo así con la solicitud del usuario.

Ejercicio 3: Despliegues

¿Cómo puedo conocer los replicasets asociados a un deployment? ¿Y los pods asociados a un deployment? Explique las diferencias entre contenedor, pod, replicaset, job y deployment.

Para saber los *replicasets* asociados a un deployment podemos ejecutar el siguiente comando: *kubectl get replicasets:*

```
NAME                       DESIRED   CURRENT   READY   AGE
frontend-9ff57fc69         3         3         3       16m
redis-follower-5bf955ffdc  2         2         2       16m
redis-leader-f8b565846     1         1         1       16m
```

Para el caso de los pods podemos ejecutar *kubectl get pods:*

```
NAME                             READY   STATUS    RESTARTS   AGE
frontend-9ff57fc69-7tb4p         1/1     Running   0          18m
frontend-9ff57fc69-c48qk         1/1     Running   0          18m
frontend-9ff57fc69-zfqhp         1/1     Running   0          18m
redis-follower-5bf955ffdc-gwfnb  1/1     Running   0          18m
redis-follower-5bf955ffdc-lsqpl  1/1     Running   0          18m
redis-leader-f8b565846-7h9zv     1/1     Running   0          18m
```

En cuanto a las diferencias entre contenedor, pod, replicaset, job y deployment:

Contenedor: Es la unidad mínima de software empaquetado (la imagen Docker). Contiene el código de la aplicación y sus librerías, pero Kubernetes no lo gestiona directamente, sino que siempre va dentro de un Pod.

Pod: Es la unidad atómica de Kubernetes. Envuelve a uno o más contenedores (normalmente uno). Comparte almacenamiento y red (IP). Es efímero: si muere, desaparece y no se reinicia solo; hace falta alguien que lo supervise.

ReplicaSet: Es el "supervisor". Su única misión es asegurar que siempre haya un número exacto de copias (réplicas) de un Pod ejecutándose. Si un Pod muere, el ReplicaSet se da cuenta y crea uno nuevo idéntico.

Deployment: Es el "gerente de alto nivel" que gestiona a los ReplicaSets. Permite hacer actualizaciones de versión (Rolling Updates) y volver atrás (Rollbacks). Tú hablas con el Deployment (cambias la imagen), el Deployment crea un nuevo ReplicaSet y apaga el viejo de forma ordenada.

Job: A diferencia de los anteriores (que están pensados para servicios que nunca paran, como una web), un Job crea Pods para realizar una tarea finita y terminar. Por ejemplo: hacer un cálculo matemático o una copia de seguridad. Cuando acaba, el Pod no se reinicia, se queda en estado "Completed".

Ejercicio 4: Servicios

Describa los tres tipos principales de servicios en Kubernetes. ¿Cuál es el tipo por defecto? ¿Qué diferencia existe entre port, targetPort y nodePort? ¿Se puede especificar el protocolo de nivel de transporte? Explique los diferentes tipos de servicios, cómo funciona MetalLB y la diferencia de un servicio y un Ingress Controller.

Existen tres tipos principales de servicios en Kubernetes:

ClusterIP: Expone el servicio a través de una dirección IP interna. Hace que el servicio solo sea accesible desde dentro del clúster. Es ideal para comunicación entre microservicios (como nuestra base de datos Redis).

NodePort: Expone el servicio en la misma IP de cada nodo seleccionado, en un puerto estático (generalmente alto, 30000-32767). Permite el acceso externo contactando a <IP-Nodo>:<NodePort>.

LoadBalancer: Crea un balanceador de carga externo (si el proveedor de nube lo soporta) y asigna una IP externa fija al servicio. Es el método estándar para exponer servicios a internet.

-El tipo por defecto es ClusterIP. Si no se especifica el type en el YAML, el servicio será privado.

-Diferencia entre port, targetPort y nodePort

port: Es el puerto en el que el Servicio (la IP virtual) escucha las peticiones. Es el puerto que usan otros pods para llamar a este servicio.

targetPort: Es el puerto del Pod (contenedor) al que el servicio reenvía el tráfico. Es donde tu aplicación realmente está escuchando (ej. puerto 80 para Nginx/PHP).

nodePort: Es el puerto que se abre físicamente en cada Nodo del clúster para permitir acceso externo. Solo se usa en servicios tipo NodePort o LoadBalancer.

-Sí, se puede especificar. El campo protocol en la definición del servicio permite elegir entre TCP (por defecto), UDP y SCTP.

-En clústeres "bare-metal" (físicos, no en la nube como AWS/Google), no existe un balanceador de carga "mágico" que te dé IPs externas. MetalLB es la pieza de software que soluciona esto. Se encarga de asignar direcciones IP de un rango reservado (en nuestro caso, la red de estudiantes 10.1.x.x) a los servicios tipo LoadBalancer, haciendo que sean accesibles desde la red del laboratorio . Funciona principalmente a nivel de capa 3 y 4.

-**Servicio** (Capa 4): Enruta tráfico basándose en IP y Puerto (TCP/UDP). Un servicio LoadBalancer suele exponer una IP para un solo servicio.

-**Ingress Controller** (Capa 7): No es un tipo de servicio, sino un enrutador inteligente (HTTP/HTTPS) que se pone "delante" de los servicios. Permite reglas avanzadas como dominios (miweb.com), rutas (/app1, /app2) y terminación SSL (candado verde). Un solo Ingress puede repartir tráfico a múltiples Servicios internos

Ejercicio 5: Tutorial II

Utilizando los ficheros proporcionados, realice un despliegue de un cluster de Apache Spark en Kubernetes. Responda a las siguientes preguntas:

- ¿Cómo usaría el cluster de Spark desde el JupyterLab? Proporcione un ejemplo de funcionamiento básico.
- Escala el número de workers, mediante `kubectl apply -f` y otro comando. Observe el cambio en Kubernetes y en la interfaz de gestión de Spark.
- Destruye el cluster de Spark, mediante `kubectl delete -f`.
- ¿Pueden dos cluster de Spark co-existir? Indique los cambios a realizar.

Empezamos por realizar el despliegue de Spark en Kubernetes a través de los yaml de las diapositivas y añadiendo limits:

Despliegue master:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spark-master
spec:
  replicas: 1
  selector:
    matchLabels:
      component: spark-master
  template:
    metadata:
      labels:
        component: spark-master
    spec:
      containers:
        - name: spark-master
          image: dperdices/spark-master:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 7077   # puerto Spark
            - containerPort: 8080   # WebUI
          resources:
            requests:
              cpu: "500m"
              memory: "512Mi"
            limits:
              cpu: "1"
              memory: "1Gi"
```

Despliegue worker:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spark-worker
spec:
  replicas: 2            # número inicial de workers
  selector:
    matchLabels:
      component: spark-worker
  template:
    metadata:
      labels:
        component: spark-worker
    spec:
      containers:
        - name: spark-worker
          image: dperdices/spark-worker:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8081   # WebUI del worker
          resources:
            requests:
              cpu: "500m"
              memory: "512Mi"
            limits:
              cpu: "1"
              memory: "1Gi"
```

Servicio master:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: spark-master
spec:
  type: LoadBalancer
  ports:
    - name: webui
      port: 8080
      targetPort: 8080
    - name: spark
      port: 7077
      targetPort: 7077
  selector:
    component: spark-master
```

Tras esto podemos ver que los pods han sido creados:

```
[cap1462_09@openhpc spark-cluster]$ kubectl get pods
kubectl get svc
NAME                            READY   STATUS    RESTARTS   AGE
spark-master-6dffd6ffb5-7b6cq   1/1     Running   0          10s
spark-worker-5568b67d45-c4p6m   1/1     Running   0          10s
spark-worker-5568b67d45-rmts5   1/1     Running   0          10s
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                         AGE
spark-master   LoadBalancer   10.43.95.192   10.162.9.1    8080:32040/TCP,7077:32584/TCP   10s
```

• ¿Cómo usaría el cluster de Spark desde el JupyterLab? Proporcione un ejemplo de funcionamiento básico.

Desde JupyterLab se usa el clúster de Spark creando una sesión de SparkSession apuntando al master de Kubernetes. En la guía del clúster nos dan el patrón a seguir, donde el master es:

SPARK_URL = "spark://spark-master.spark.svc.cluster.local:7077"

y se construiría así la sesión Spark:

```
# Dirección del master del cluster Kubernetes
SPARK_URL = "spark://spark-master.spark.svc.cluster.local:7077"
```

```
IP = os.popen("hostname -I").read().strip().split()[0]

spark = (SparkSession.builder
        .appName("Spark Demo")
        .master(SPARK_URL)
        .config("spark.driver.host", IP)
        .config("spark.executor.memory", "512M")
        .config("spark.cores.max", 2)
        .config("spark.executor.instances", 2)
        .getOrCreate())

sc = spark.sparkContext
```

Ejecutando este código, ya tendremos una sesión de Spark desplegada en Kubernetes y podremos ejecutar trabajos de forma distribuida.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
result = rdd.map(lambda x: x * 2).collect()

print("Resultado:", result)

spark.stop()
```

Finalmente, ejecutamos este pequeño programa de ejemplo que realiza la multiplicación de 5 números en paralelo.

```
(base) jovyan@jupyter-cap1462-09---352a3d8d:~/Desktop$ python3 spark-demo.py
WARNING: Using incubator modules: jdk.incubator.vector
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/11/24 18:14:42 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Resultado: [2, 4, 6, 8, 10]
```

Como vemos funciona perfectamente.

• Escala el número de workers, mediante kubectl apply -f y otro comando. Observe el cambio.

Vamos a empezar escalando el número de workers subiendo el número de réplicas a 4 en las spec del fichero de despliegue spark-worker-deployment.yaml.

```
[cap1462_09@openhpc spark-cluster]$ vi spark-worker-deployment.yaml
[cap1462_09@openhpc spark-cluster]$ kubectl apply -f spark-worker-deployment.yaml
deployment.apps/spark-worker configured
[cap1462_09@openhpc spark-cluster]$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
spark-master-6dffd6ffb5-7b6cq   1/1     Running   0          28m
spark-worker-5568b67d45-7p55b   1/1     Running   0          10s
spark-worker-5568b67d45-c4p6m   1/1     Running   0          28m
spark-worker-5568b67d45-rmts5   1/1     Running   0          28m
```

Tras realizar el despliegue y ver los pods, vemos que solo se han creado 3 workers cuando hemos pedido 4, esto se debe a que nuestro cluster tiene como máximo 4GiB de memoria, lo cual excedería si se crease un cuarto worker.

Ahora probraremos a escalar con el comando **kubectl scale:**

```
[cap1462_09@openhpc spark-cluster]$ kubectl scale --replicas=4 deployment/spark-worker
deployment.apps/spark-worker scaled
[cap1462_09@openhpc spark-cluster]$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
spark-master-6dffd6ffb5-7b6cq   1/1     Running   0          34m
spark-worker-5568b67d45-7p55b   1/1     Running   0          6m33s
spark-worker-5568b67d45-c4p6m   1/1     Running   0          34m
spark-worker-5568b67d45-rmts5   1/1     Running   0          34m
```

Este comando modifica el campo replicas del deployment en el servidor API y Kubernetes vuelve a crear/eliminar pods hasta alcanzar ese número.

• Destruye el cluster de Spark, mediante kubectl delete -f.

Para ello hacemos uso del comando kubecetl delete -f y borramos los despliegues y servicios:

```
[cap1462_09@openhpc spark-cluster]$ kubectl delete -f spark-master-deployment.yaml
kubectl delete -f spark-worker-deployment.yaml
kubectl delete -f spark-master-service.yaml
deployment.apps "spark-master" deleted from cap1462-09 namespace
deployment.apps "spark-worker" deleted from cap1462-09 namespace
service "spark-master" deleted from cap1462-09 namespace
```

• ¿Pueden dos cluster de Spark co-existir? Indique los cambios a realizar.

Sí pueden coexistir siempre y cuando se diferencien por nombres, labels y servicios.
Es decir, si queremos crear otro cluster Spark tenemos que tener unos deployments y servicios con nombres distintos:

- Cluster 1: spark-master-a, spark-worker-a, spark-master-service-a
- Cluster 2: spark-master-b, spark-worker-b, spark-master-service-b

En cuanto a los services, cada uno debe tener su propia CLUSTER–IP y en el caso de usar LoadBalancer su propia EXTERNAL-IP.
Finalmente deben utilizar URLs diferentes al conectarse desde JupyterLab de modo que la aplicación Spark se ejecute contra el clúster correspondiente.

Ejercicio 6: Estado

¿Puedo especificar el estado (status) de un objeto mediante `kubectl apply`?

No, no es posible especificar ni modificar el campo status de un objeto de Kubernetes usando kubectl apply. Esto se debe a que el campo status representa información interna generada por el sistema, donde Kubernetes describe cuál es el estado real del objeto en ese momento (pods creados, IP asignada, condiciones, eventos, etc.).

El usuario solo puede definir la parte spec, que es la descripción deseada del objeto. A partir de esa especificación, es el propio plano de control de Kubernetes quien calcula y actualiza automáticamente el campo status para reflejar la situación actual del recurso.

Ejercicio 7: Metadatos y etiquetas

Los metadatos son uno de los atributos más importantes de Kubernetes que controlan como despliegues, templates y servicios interactúan. Compruebe (o refute) que puede poner dentro de labels el nombre de atributo y valor de atributo que desee. Explique como realizar una busqueda de pods según el valor de las etiquetas.

En Kubernetes es posible asignar prácticamente cualquier combinación de clave y valor dentro del apartado metadata.labels. Las etiquetas no están limitadas a un conjunto predefinido, sino que pueden utilizarse libremente para clasificar y organizar recursos según las necesidades del usuario. La única restricción es cumplir el formato válido de claves y valores establecido por Kubernetes (caracteres alfanuméricos, puntos y guiones), pero más allá de esas reglas sintácticas no existe limitación semántica. Esto permite crear etiquetas personalizadas para identificar despliegues, versiones, propietarios o cualquier otro criterio que resulte útil dentro del entorno de trabajo.

Las etiquetas cumplen una función esencial en la interacción entre objetos de Kubernetes. Servicios, despliegues y replicasets utilizan selectors basados en etiquetas para determinar qué pods deben gestionar. Gracias a ello, la comunicación entre componentes se mantiene desacoplada y flexible: basta con modificar una etiqueta para que un pod pase a formar parte de un servicio diferente o de un conjunto alternativo de réplicas. Las etiquetas son, por tanto, uno de los mecanismos centrales para estructurar las aplicaciones dentro del clúster.

Además de su papel en la lógica interna de Kubernetes, las etiquetas permiten realizar búsquedas precisas de recursos. El comando kubectl facilita esta operación mediante el parámetro -l, que selecciona únicamente los objetos que coinciden con una determinada clave y valor.

Si probamos a ejecutar kubectl get pods -l component=spark-worker:

```
[cap1462_09@openhpc spark-cluster]$ kubectl get pods -l component=spark-worker
NAME                            READY   STATUS    RESTARTS   AGE
spark-worker-5568b67d45-fxtzd   1/1     Running   0          21m
spark-worker-5568b67d45-qnvtl   1/1     Running   0          21m
spark-worker-5568b67d45-rfkcg   1/1     Running   0          21m
```

Como vemos esto muestra solo los pods etiquetados como workers dentro del clúster de Spark. Esta capacidad de filtrado es fundamental para administrar recursos de forma eficiente, especialmente en entornos donde conviven múltiples servicios y despliegues.

Ejercicio 8: Alta disponibilidad de despliegues

Una característica interesante de los deployments es la posibilidad de actualizarlo con el menor tiempo de no disponibilidad posible. Esto lo gestiona automáticamente Kubernetes, ¿cómo lo hace para el caso del que cambiamos la imagen del contenedor? Mediante el comando `kubectl set image` pruebe a actualizar una imagen de un despliegue y observe los diferentes replicasets y pods y explique lo que sucede.

Al actualizar la imagen del Deployment mediante el comando kubectl set image, Kubernetes crea automáticamente un nuevo ReplicaSet asociado a la imagen modificada. En nuestro caso, el Deployment spark-master pasó de tener un único ReplicaSet (spark-master-6dffd6ffb5) a disponer de un segundo ReplicaSet recién creado (spark-master-854566cb74).

```
[cap1462_09@openhpc spark-cluster]$ kubectl get replicasets
NAME                      DESIRED   CURRENT   READY   AGE
spark-master-6dffd6ffb5   1         1         1       69m
spark-master-854566cb74   1         0         0       2m20s
spark-worker-5568b67d45   2         2         2       69m
```

El ReplicaSet antiguo mantiene las réplicas activas, mientras que el nuevo aparece inicialmente con CURRENT=0 y READY=0. Esto ocurre porque Kubernetes intenta iniciar un nuevo pod con la imagen actualizada, pero no puede programarlo hasta que existan recursos suficientes en el namespace.

Este comportamiento demuestra el funcionamiento del mecanismo de rolling update: el sistema prepara un nuevo ReplicaSet, pretende escalarlo gradualmente y, una vez que el nuevo pod se encuentre en estado Running, reducirá las réplicas del ReplicaSet antiguo. Debido a las limitaciones de recursos (4 cores y 4 GiB por usuario), el nuevo pod no puede crearse de inmediato, lo que se refleja en que el ReplicaSet nuevo existe pero no tiene réplicas activas. Cuando se liberan recursos, Kubernetes completa la actualización sin interrumpir el servicio, manteniendo la disponibilidad del clúster en todo momento.