

AUTOESCALADO EN UN CLUSTER DE KUBERNETES CON KEDA



Fabio Gonzalez del Valle | Trabajo de Fin de Grado

2º Administración de Sistemas Informáticos en Red

IES Gonzalo Nazareno

2023/2024

- Índice -

1. Objetivos que se quieren conseguir y se han conseguido.....	3
2. Escenario necesario para la realización del proyecto.....	4
3. Fundamentos teóricos y conceptos.....	5
3.1. - Keda (Kubernetes Event-driven Autoscaler).....	5
3.1.1 - ¿Como funciona Keda?.....	5
3.1.2 - ¿Por qué usar Keda?.....	6
3.2. - AlertManager.....	7
3.2.1 - ¿Como funciona AlertManager?.....	7
3.2.2 - Funciones principales de AlertManager.....	7
3.3 - Prometheus.....	8
3.3.1 - ¿Cómo funciona Prometheus?.....	8
3.3.2 - ¿Por qué usar Prometheus?.....	9
3.4. - RabbitMQ.....	10
3.4.1 Descripción de RabbitMQ como escalador en KEDA.....	10
3.5 - Grafana.....	11
3.5.1 Grafana en conjunto con Prometheus.....	11
3.6 - Helm.....	12
3.6.1 Componentes Principales de Helm.....	12
4. Descripción detallada de lo que se ha realizado.....	13
4.1 – Instalación y configuración de Docker, Kind y Kubectl.....	13
4.2 - Creación del escenario.....	16
4.2 - Configuración y despliegue de las aplicaciones para la demostración.....	18
4.2.1 - Demostración con MongoDB.....	19
4.2.2 - Demostración con Redis.....	27
4.2.3 - Demostración con RabbitMQ.....	31
4.3 - Comparativa entre Keda y HPA.....	34
4.3.1 - Principales diferencias entre HPA y Keda.....	35
5. Conclusiones y propuestas para seguir trabajando sobre el tema.....	37
6. Dificultades que se han encontrado (optativo).....	38
7. Bibliografía.....	40

1. Objetivos que se quieren conseguir y se han conseguido.

Después de terminar el ciclo formativo y de haber creado y trabajado con diferentes entornos, he decidido enfocar mi proyecto en un tema relacionado con Kubernetes, ya que actualmente es la tecnología más utilizada para la orquestación de contenedores. Considero que es un tema muy interesante y algo con lo que me gustaría trabajar en el futuro. Kubernetes ofrece una gran facilidad para crear y desplegar aplicaciones, así como para implementar el despliegue continuo de estas.

De aquí surge la idea de proyecto, la cual consiste principalmente en implementar Keda (Kubernetes Event-driven Autoscaler) sobre un clúster de Kubernetes en local creado desde 0. Con este escenario, se quiere conseguir la automatización y autoescalado de recursos del clúster mediante métricas proporcionadas por otro servidor.

Objetivos propuestos y conseguidos:

- Creación de un clúster de Kubernetes sobre contenedores Docker en local usando Kind, con varios control-planes y varios nodos workers
- Implementación de un servidor de métricas como Prometheus, necesario para poder realizar el autoescalado parametrizado con Keda.
- Pruebas de autoescalado usando diferentes proveedores de métricas interpretadas por Keda, como pueden ser, RabbitMQ, MongoDB y Redis entre otros.
- Comparativa entre Keda y la principal solución ofrecida por Kubernetes (HPA)

2. Escenario necesario para la realización del proyecto.

Para la realización del proyecto, he trabajado sobre un escenario el cual esta compuesto por las siguientes tecnologías:

- Docker, necesario para crear el clúster, aunque también podemos hacer uso de máquinas virtuales
- Kind, para desplegar el clúster sobre los contenedores docker
- Kubectl, debe estar instalado en nuestro equipo local para poder interactuar con la API de Kubernetes
- Un sistema de monitorización como Prometheus, para que Keda pueda interpretar las métricas y usarlas en el autoescalado.
- Un balanceador de carga, como Metallb, para que los servicios sean accesibles desde fuera del clúster.
- Keda, que se encargará del autoescalado horizontal de los pods dependiendo de diferentes parámetros
- Un sistema de alertas en tiempo real, como AlertManager
- (Opcional) Podemos añadir Grafana al cluster, para el visionado de métricas provistas por Prometheus
- Fuentes de datos externas para que Keda pueda ejecutar el autoescalado, en este caso he usado MongoDB, RabbitMQ y Redis

3. Fundamentos teóricos y conceptos

En este apartado, voy a explicar en profundidad para que sirven cada uno de los componentes que conforman el escenario y como se complementan entre ellos, conformando así un sistema de monitorización, autoescalado automático y alertas en tiempo real en el cluster.

3.1. - Keda (Kubernetes Event-driven Autoscaler)

Keda (Kubernetes-based Event-Driven Autoscaler) es una herramienta que se integra con Kubernetes para proporcionar escalado automático basado en eventos. Es especialmente útil en entornos donde las cargas de trabajo son impulsadas por eventos, como colas de mensajes, sistemas de transmisión o eventos de Kafka. Es un proyecto de CNCF (Cloud Native Computing Foundation) en fase de graduación.

3.1.1 - ¿Como funciona Keda?

KEDA funciona a través de métricas externas, como colas de mensajes, sistemas de almacenamiento de datos o incluso otras cargas de trabajo de Kubernetes. A medida que se generan eventos, KEDA escala automáticamente las aplicaciones en ejecución en el clúster para procesar la carga de trabajo.

Esto significa que puedes ir más allá de las métricas estándar de CPU, memoria o uso de red y definir métricas específicas que representen mejor la carga de trabajo de tu aplicación.

A continuación veremos una tabla comparativa entre Keda y HPA (Horizontal Pod Autoscaler), la solución que nos ofrece Kubernetes para el autoescalado:

Característica	KEDA	HPA
Mecanismo de escalado	Eventos	Métricas de recursos (CPU, memoria, red)
Fuentes de eventos	Colas de mensajes, sistemas de almacenamiento de datos, trabajos de Kubernetes, métricas personalizadas	Métricas de recursos de Kubernetes
Escalabilidad	Altamente escalable	Altamente escalable
Complejidad	Más complejo de configurar que HPA	Más simple de configurar que KEDA
Casos de uso ideales	Aplicaciones basadas en eventos, aplicaciones con métricas personalizadas	Aplicaciones sin estado, aplicaciones con métricas de recursos estándar

3.1.2 - ¿Por qué usar Keda?

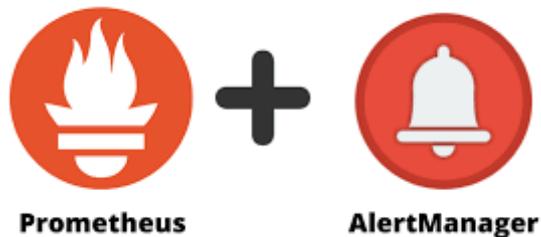
Hay varias razones para considerar el uso de Keda en entornos basados en Kubernetes:

- Escalado eficiente basado en eventos: Keda permite escalar automáticamente los pods en función de eventos específicos, lo que garantiza que los recursos se asignen solo cuando sea necesario.
- Soporte para una amplia gama de fuentes de eventos: Keda es compatible con una variedad de fuentes de eventos, como: InfluxDB, Loki, MySQL, OpenStack Metric, Apache Kafka, entre otros.
- Configuración flexible: Keda ofrece una configuración flexible que permite a los usuarios definir reglas de escala personalizadas en función de eventos específicos o métricas de recursos de Kubernetes. Esto permite adaptar el comportamiento de escalado según las necesidades específicas de la aplicación.

Para obtener más información sobre Keda: <https://keda.sh/docs/2.14/concepts/>

3.2. - AlertManager

AlertManager es un gestor de alertas para clientes como Prometheus, capaz de gestionarlas, personalizarlas y notificarlas a través de diferentes medios como Slack, Correo electrónico, OpsGenie, WeChat o Telegram.



3.2.1 - ¿Como funciona AlertManager?

En un entorno Kubernetes, Alertmanager se implementa como un pod independiente y se integra con Prometheus mediante una API REST. Cuando Prometheus detecta una alerta, la envía a Alertmanager para su procesamiento.

3.2.2 - Funciones principales de AlertManager

Entre las funciones principales de AlertManager, se encuentran:

- Elimina alertas duplicadas para evitar la saturación de notificaciones.
- Agrupa alertas relacionadas en una sola notificación para facilitar su gestión.
- Envía alertas a los canales de notificación adecuados, como correo electrónico, SMS, PagerDuty o Slack.
- Silencia las alertas de forma temporal o permanente para evitar notificaciones innecesarias.
- Se integra con diversas herramientas de monitoreo y notificación de terceros.

Para obtener más información: <https://prometheus.io/docs/alerting/latest/alertmanager/>

3.3 - Prometheus

Prometheus es una herramienta de monitoreo y alerta diseñada para la recolección y consulta de métricas. Forma parte de la Cloud Native Computing Foundation (CNCF) y es una solución estándar para el monitoreo de aplicaciones en contenedores y entornos de microservicios. Prometheus es especialmente conocido por su capacidad de escalar y manejar grandes volúmenes de datos de métricas en tiempo real.

3.3.1 - ¿Cómo funciona Prometheus?

Prometheus opera mediante un modelo de extracción (pull), en el que periódicamente recopila datos de métricas desde los endpoints configurados. Estos endpoints exponen métricas en un formato específico que Prometheus puede leer y almacenar en su base de datos de series temporales. Las principales características de su funcionamiento incluyen:

- **Modelo de Datos de Series Temporales:** Las métricas se almacenan como series temporales, identificadas por un nombre de métrica y pares de etiquetas (key-value).
- **Lenguaje de Consulta PromQL:** Para consultar y analizar los datos de métricas, Prometheus utiliza su propio lenguaje de consulta, PromQL, que permite realizar consultas complejas sobre los datos almacenados.
- **Alertas Basadas en Reglas:** Prometheus permite definir reglas de alerta basadas en las métricas recolectadas. Estas alertas pueden ser enviadas a diversos sistemas de notificación a través del componente Alertmanager.
- **Exporters:** Los exporters son componentes que recogen métricas de sistemas y servicios externos y las exponen en un formato que Prometheus pueda recolectar. Existen exporters para una amplia variedad de aplicaciones y servicios.

A continuación vamos a ver una tabla comparativa entre Prometheus y otros sistemas de monitorización como Grafana o ELK Stack:

Característica	Prometheus	Grafana	ELK Stack
Modelo de Datos	Series Temporales	Visualización (utiliza Prometheus u otros)	Logs y Series Temporales
Lenguaje de Consulta	PromQL	No tiene propio (utiliza PromQL u otros)	Elasticsearch Query DSL
Alertas	Integrado con AlertManager	No propio (usa Prometheus/Alertmanager)	Basado en Elastic Stack
Escalabilidad	Alta	Depende de la fuente de datos	Alta
Integración	Amplia con Kubernetes y CNCF	Amplia (especialmente con Prometheus)	Integración con el ecosistema Elastic
Caso de Uso	Monitoreo de métricas en tiempo real	Visualización de datos de monitoreo	Análisis de logs y métricas

3.3.2 - ¿Por qué usar Prometheus?

Prometheus es una de tantas soluciones que nos ofrece Kubernetes para la monitorización del cluster, en mi caso, he decidido usar Prometheus por la alta compatibilidad que tiene con Keda y AlertManager, ya que quedará obteniendo las métricas necesarias de Prometheus, entre otros servicios.

Para obtener más información: <https://prometheus.io/docs/introduction/overview/>

3.4. - RabbitMQ

RabbitMQ es un sistema de mensajería y manejo de cola de mensajes. Se utiliza para manejar la mensajería asincrónica entre sistemas distribuidos, permitiendo que las aplicaciones se comuniquen entre sí de manera confiable y escalable.



3.4.1 Descripción de RabbitMQ como escalador en KEDA

RabbitMQ como escalador en KEDA implica usar las métricas de RabbitMQ (como la longitud de las colas) para desencadenar el autoescalado de los pods en Kubernetes. Cuando la longitud de una cola supera un umbral definido, KEDA escala el número de réplicas de la aplicación para manejar la carga adicional.

Configurar y gestionar KEDA y RabbitMQ puede ser complejo, ya que a diferencia de otros escaladores, RabbitMQ requiere configurar una serie de secretos y políticas de seguridad para proporcionar conexión a Keda.

Para más información: <https://www.rabbitmq.com/docs>

3.5 - Grafana

Grafana es una herramienta esencial para el monitoreo y la visualización de métricas en un clúster de Kubernetes, este se utiliza para supervisar el rendimiento y el estado de los componentes del clúster, las aplicaciones desplegadas y la infraestructura.

3.5.1 Grafana en conjunto con Prometheus

Prometheus despliega un componente llamado Prometheus server que recopila métricas de varios exporters. Los exporters son agentes que se ejecutan en los nodos del clúster, dentro de los pods o como servicios, y exponen métricas en formato que Prometheus puede recolectar.

Prometheus obtiene métricas mediante una técnica llamada scraping, que consiste en hacer peticiones HTTP a endpoints de métricas expuestos por los exporters.

Prometheus tiene configuraciones de scraping definidas en su archivo de configuración (prometheus.yaml), donde se especifican los endpoints y la frecuencia de scraping.

Consulta de Métricas:

- Grafana se conecta a Prometheus como una fuente de datos.
- Los usuarios pueden escribir consultas en el lenguaje de consulta de Prometheus, para obtener métricas específicas y visualizarlas en los paneles de Grafana.

Visualización:

- Grafana permite crear paneles de control (dashboards) que consisten en múltiples paneles (graphs, tables, singlestat, etc.).
- Cada panel se configura con una consulta PromQL para mostrar los datos deseados.
- Grafana permite la visualización en tiempo real, con actualizaciones automáticas de los gráficos y paneles.

3.6 - Helm

Helm es una herramienta de gestión de paquetes para Kubernetes, la cual nos posibilita la instalación o configuración de las aplicaciones de nuestro clúster. Funciona de manera similar a otros gestores de paquetes como apt o yum.

Aunque se trata de una herramienta básica en cualquier clúster de Kubernetes, he visto necesario que hablemos de ella ya que será indispensable para el montaje y demostración del escenario.

3.6.1 Componentes Principales de Helm

Helm CLI (Command Line Interface):

La herramienta de línea de comandos que los usuarios interactúan directamente para gestionar las aplicaciones.

Charts:

Un chart es un paquete que contiene toda la información necesaria para crear una instancia de una aplicación en Kubernetes. Incluye templates de Kubernetes YAML, valores de configuración, archivos de dependencias, y más.

Helm Repositories:

Un repositorio de charts es una colección de charts empaquetados y versionados, disponibles para descarga y uso.

Releases:

Una release es una instancia específica de un chart desplegado en un clúster de Kubernetes. Puedes tener múltiples releases de un mismo chart, cada una configurada de manera diferente.

4. Descripción detallada de lo que se ha realizado

En este apartado, voy a explicar paso por paso todo el procedimiento que se ha realizado para llegar a tener el escenario y poder realizar la demostración.

4.1 – Instalación y configuración de Docker, Kind y Kubectl.

Docker:

1. Preparamos el sistema actualizando la paquetería:

```
sudo apt-get update
sudo apt-get upgrade
```

2. Eliminamos cualquier rastro de Docker, esto lo haremos para eliminar cualquier software relacionado con Docker que pueda causar conflictos:

```
sudo apt remove docker-desktop
rm -r $HOME/.docker/desktop
sudo rm /usr/local/bin/com.docker.cli
sudo apt purge docker-desktop
```

3. Instalamos el repositorio de Docker

```
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

También será necesario añadir a nuestro sistema la clave GPG de Docker:

```
sudo mkdir -p /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg

-----

echo \

  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \

  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

4. Instalamos el motor de Docker.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

5. Podemos comprobar la instalación ejecutando el siguiente comando:

```
sudo docker run hello-world
```

6. Para ejecutar Docker sin necesidad de ser usuario root:

```
sudo groupadd docker

sudo usermod -aG docker $USER
```

Kind:

Kind "Kubernetes IN Docker," es una herramienta diseñada para ejecutar clústeres de Kubernetes dentro de contenedores Docker. Con esta podremos crear entornos de pruebas de forma rápida y sencilla sin necesitar muchos recursos.

1. Descargamos el binario de kind:

```
[ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.23.0/kind-linux-amd64
```

2. Damos permisos y movemos al directorio donde se encuentran todos los binarios:

```
chmod +x ./kind  
sudo mv ./kind /usr/local/bin/kind
```

Kubectl:

Kubectl es el comando que nos permitirá interactuar con nuestro clúster de Kubernetes.

1. Descargamos la última versión:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

2. Una vez descargado, procedemos a la instalación del binario:

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

3. Podemos comprobar que está correctamente instalado ejecutando el comando:

```
kubectl version --client
```

4.2 - Creación del escenario

Una vez hemos descargado e instalado las tres herramientas anteriores tenemos todo lo necesario para poder crear el escenario. Tenemos diferentes formas de configurarlo, podemos disponer de varios control-planes y varios nodos workers trabajando en clúster, en mi caso he optado por crear un escenario simple, el cual consiste en un control-plane y dos nodos workers.

Al ser un escenario sencillo, se optimiza el consumo de recursos, aunque el escenario también podría constar de varios control-planes y nodos workers, o un solo nodo control-plane, la demostración funcionaría en todos los casos.

Mi forma de crear el clúster ha sido mediante un fichero `.yaml`, el cual contiene toda la configuración del mismo y un Makefile mediante el cual podemos elegir entre diferentes opciones para instalar o desinstalar el cluster, también tenemos la opción de instalar Metallb como balanceador de carga.

`kind-cluster.yaml`

```
! kind-cluster.yaml
1  kind: Cluster
2  apiVersion: kind.x-k8s.io/v1alpha4
3  nodes:
4  - role: control-plane
5    kubeadmConfigPatches:
6    - |
7      kind: InitConfiguration
8      nodeRegistration:
9        kubeletExtraArgs:
10         node-labels: "ingress-ready=true"
11    extraPortMappings:
12    - containerPort: 80
13      hostPort: 80
14      protocol: TCP
15    - containerPort: 443
16      hostPort: 443
17      protocol: TCP
18  - role: worker
19  - role: worker
20
```

Makefile

```

M Makefile
1  install:
2      echo "Configurando el cluster"
3      kind create cluster --name kind-keda \
4          --config kind-cluster.yaml
5  lb:
6      echo "Configurando metallb"
7      metallb/metallb.sh
8  delete:
9      echo "Eliminando el cluster"
10     kind delete cluster --name kind-keda
11  all: install lb
12

```

De esta forma tenemos la capacidad de crear o eliminar de manera rápida y sencilla nuestro clúster, con la línea:

```
node-labels: "ingress-ready=true"
```

Hemos habilitado la opción de usar Ingress Controllers en todos los nodos de nuestro clúster, también hemos mapeado los puertos 80 y 443.

Si hemos realizado estos pasos correctamente tendremos lo siguiente:

```

fabio@Fabio-PC ~/$ kind kubectx --name kind-keda
fabio@Fabio-PC ~/$ kubectl get nodes -o wide
NAME                STATUS    ROLES    AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION   CONTAINER-RUNTIME
kind-keda-control-plane Ready    control-plane 3h7m   v1.30.0   172.18.0.4    <none>        Debian GNU/Linux 12 (bookworm) 6.6.22-linuxkit  containerd://1.7.15
kind-keda-worker    Ready    <none>     3h7m   v1.30.0   172.18.0.2    <none>        Debian GNU/Linux 12 (bookworm) 6.6.22-linuxkit  containerd://1.7.15
kind-keda-worker2   Ready    <none>     3h7m   v1.30.0   172.18.0.3    <none>        Debian GNU/Linux 12 (bookworm) 6.6.22-linuxkit  containerd://1.7.15

```

Y los contenedores Docker:

Name	Image	Status	Port(s)
 kind-keda-worker2 8f26cd0a6fa8	kindest/node:v1.30.0	Running	
 kind-keda-control-plane bead7268e528	kindest/node:v1.30.0	Running	443:443 42103:6443 80:80 Show less
 kind-keda-worker d398931c7012	kindest/node:v1.30.0	Running	

4.2 - Configuración y despliegue de las aplicaciones para la demostración.

En este apartado voy a explicar la parte de las aplicaciones que voy a desplegar para demostrar el funcionamiento de Keda en el clúster de Kubernetes.

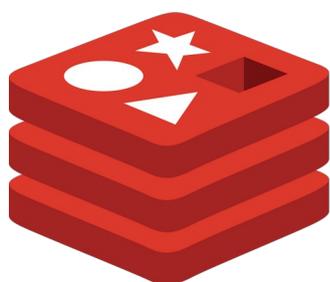
Para ello he preparado 3 aplicaciones:

- MongoDB → Esta es una aplicación simple, Keda usa las consultas insertadas en los pods de Mongo para hacer el escalado.
- Redis → Con esta aplicación, Keda hará el autoescalado basado en una cola de Redis
- RabbitMQ → En esta aplicación, Keda usará una cola de RabbitMQ para el autoescalado horizontal

A continuación, pasaré a explicar cada una de estas aplicaciones de forma detallada, también dejaré el código necesario para poder desplegarlas, mi repositorio se encontrará al final del documento.



mongoDB



redis

4.2.1 - Demostración con MongoDB

Para la primera demostración he utilizado una aplicación basada en MongoDB.

app1-deployment.yaml

Este archivo define el servicio y el despliegue del balanceador de carga para los pods

```
apiVersion: v1
kind: Service
metadata:
  name: service
  labels:
    app: service
spec:
  ports:
    - port: 3232
  selector:
    app: service
    tier: frontend
  type: LoadBalancer
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: despliegue-service
  labels:
    app: service
spec:
  selector:
    matchLabels:
      app: service
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: service
        tier: frontend
    spec:
      containers:
        - name: contenedor-principal
          image: docker.io/fabiogonzalez8/mongoscaleserver:latest2
          imagePullPolicy: Always
          args:
            - app
          ports:
            - containerPort: 3232
              name: service
          resources:
            limits:
              cpu: "0.3"
```

app1-mongo-deployment.yaml

Este archivo define el service, el volumen persistente, el secret y el despliegue para MongoDB.

```
---
apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  ports:
    - port: 27017
      targetPort: 27017
  type: LoadBalancer
  selector:
    app: mongo
```

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pv-claim
  labels:
    app: mongo
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

```
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
type: Opaque
data:
  mongo-root-password: YWRtaW4xMjM=
  mongo-root-username: YWRtaW4=
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo
  labels:
    app: mongo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      dnsPolicy: ClusterFirst
      containers:
        - image: mongo
          name: mongo
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongo-persistent-storage
              mountPath: /data/db/mongo
          env:
            - name: MONGO_INITDB_ROOT_USERNAME
              valueFrom:
                secretKeyRef:
                  name: mongo-secret
                  key: mongo-root-username
            - name: MONGO_INITDB_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mongo-secret
                  key: mongo-root-password
      volumes:
        - name: mongo-persistent-storage
          persistentVolumeClaim:
            claimName: mongo-pv-claim
```

- Replicas: inicialmente 1
- Contenedor: imagen oficial de MongoDB
- Volumen persistente: Monta el volumen en /data/db/mongo
- Variables de Entorno: Usa las credenciales del secreto mongo-secret

dummy-pods.yaml

Este archivo define el ScaledObject de KEDA y la autenticación necesaria para escalar dummy-mongo basado en las métricas de MongoDB.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dummy-mongo
  labels:
    app: dummy-mongo
spec:
  replicas: 0
  selector:
    matchLabels:
      app: dummy
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: dummy
    spec:
      terminationGracePeriodSeconds: 5
      containers:
        - name: dummy
          image: docker.io/karthik3030/kedadummyserver:latest
          imagePullPolicy: IfNotPresent
          lifecycle:
            preStop:
              exec:
                command:
                  - pkill -f "sleep"
```

mongo-scaledobject.yaml

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: mongo-scaledobject
  namespace: default
  labels:
    deploymentName: dummy-mongo
spec:
  maxReplicaCount: 20
  pollingInterval: 5
  cooldownPeriod: 5
  scaleTargetRef:
    name: dummy-mongo
  triggers:
    - type: mongodb
      metadata:
        host: "10.1.1.1"
        port: "27017"
        dbName: "admin"
        collection: "scalecollection"
        query: '{"name":"test"}'
        queryValue: "1"
        activationQueryValue: "0"
      authenticationRef:
        name: mongodb-local-trigger
```

Máximo de Réplicas: Hasta 20 réplicas.

Intervalo de Sondeo: Cada 5 segundos.

Tiempo que para escalar la app hacia abajo: 5 segundos.

Despliegue que se escalará: dummy-mongo.

```
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: mongodb-local-trigger
spec:
  secretTargetRef:
    - parameter: connectionString
      name: mongodb-local-secret
      key: connect
```

```
apiVersion: v1
kind: Secret
metadata:
  name: mongodb-local-secret
type: Opaque
data:
  connect: bW9uZ29kYjovL2FkbWluOmfkbWluMTIzQG1vbmdvLmRlZmF1bHQuc3ZjLm
```

4.2.2 - Demostración con Redis

La siguiente aplicación que veremos consiste en escalar automáticamente el número de réplicas de un contenedor Nginx en función de la longitud de una lista en Redis.

app2-deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: app1-redis
5  spec:
6    selector:
7      matchLabels:
8        app: nginx
9    #replicas: 2
10   template:
11     metadata:
12       labels:
13         app: nginx
14     spec:
15       containers:
16         - name: nginx
17           image: nginx:latest
18           ports:
19             - containerPort: 80
```

app2-scaleObject.yaml

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: app1-redis
spec:
  scaleTargetRef:
    name: "app1-redis"
  pollingInterval: 5
  cooldownPeriod: 10
  idleReplicaCount: 0
  minReplicaCount: 1
  maxReplicaCount: 100
  #####
  #Este parámetro indica el número máximo de veces que
  #KEDA permitirá que falle el cálculo de la métrica de escalado
  #Si eso ocurre, las replicas pasarán a ser 6
  fallback:
    failureThreshold: 3
    replicas: 6
  #####
  advanced:
    restoreToOriginalReplicaCount: false
  triggers:
  - type: redis
    metadata:
      address: redis-master.app1.svc:6379
      listName: mylist
      listLength: "1"
      activationListLength: "1"
      enableTLS: "false"
      databaseIndex: "0"
    authenticationRef:
      name: app1-redis
```

secret-redis.yaml

```
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: app1-redis
spec:
  secretTargetRef:
    - parameter: password
      name: redis
      key: redis-password
```

kustomization.yaml

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

namespace: app2

resources:
  - app2-deployment.yaml
  - app2-scaleObject.yaml
  - secret-redis.yaml
```

Especifica los recursos (app2-deployment.yaml, app2-scaleObject.yaml, secret-redis.yaml) que se gestionarán.

Por otro lado contamos con el Chart de Helm para Redis.

Un chart de Helm es un paquete de Kubernetes que contiene una o más definiciones de recursos, como despliegues, servicios y configuraciones, empaquetados junto con plantillas de configuración y valores predeterminados. Estos charts son creados utilizando la herramienta Helm, que facilita la gestión y la instalación de aplicaciones en un clúster de Kubernetes.

En este caso, el chart proporciona los recursos necesarios para desplegar una instancia de Redis en el clúster de Kubernetes.

```
apiVersion: v2
name: redis
description: Chart de Helm para el despliegue de Redis en Kubernetes
type: application
version: 0.1.0

dependencies:
- name: redis
  version: 17.6.0
  repository: https://charts.bitnami.com/bitnami
```

```
dependencies:
- name: redis
  repository: https://charts.bitnami.com/bitnami
  version: 17.6.0
digest: sha256:e87a0d1e797059b90ff4d15523def09be2508e18943dfa56ba1b84
generated: "2024-06-03T20:34:35.281118648+02:00"
```

4.2.3 - Demostración con RabbitMQ

Despliegue de RabbitMQ: Se utiliza Helm para una instalación sencilla y rápida de RabbitMQ, con credenciales configuradas.

Publicación de Mensajes: Un Job de Kubernetes se encarga de enviar 300 mensajes a la cola RabbitMQ, lo que permitirá probar el funcionamiento del consumidor y el escalado automático.

Consumo de Mensajes: Un Deployment define un pod consumidor que recibe mensajes de RabbitMQ.

Escalado Automático con Keda: Keda está configurado para escalar automáticamente el número de pods consumidores basado en la longitud de la cola RabbitMQ.

En este caso no será necesario insertar nada manualmente para demostrar el autoescalado. Simplemente podemos cambiar los parámetros para ajustar el número de pods y mensajes.

deploy-rabbitmq.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: publicador-rabbitmq
spec:
  template:
    spec:
      containers:
        - name: cliente-rabbitmq
          image: ghcr.io/kedacore/rabbitmq-client:v1.0
          imagePullPolicy: Always
          command:
            [
              "send", # Comando para enviar mensajes
              "amqp://user:PASSWORD@rabbitmq.default.svc.cluster.local:5672", # URL de conexión a RabbitMQ
              "300", # Número de mensajes a enviar
            ]
          restartPolicy: Never # Política de reinicio del contenedor
      backoffLimit: 4 # Límite de reintentos en caso de fallo
```

Este recurso define un Job de Kubernetes que utiliza la imagen `ghcr.io/kedacore/rabbitmq-client:v1.0` para publicar 300 mensajes en la cola RabbitMQ. La imagen es una oficial sacada del repositorio de Keda en Github.

deploy-trigger.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: secreto-consumidor-rabbitmq
data:
  RabbitMqHost: YW1xcDovL3VzZXI6UEFTU1dPUkRAcmFiYm10bXEuZGVmYXVsdC5zdmMuY2x1c3Rlci5sb2NhbDo1Njcy
```

Este Secret almacena la URL de conexión a RabbitMQ codificada en base64.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: consumidor-rabbitmq
  namespace: default
  labels:
    app: consumidor-rabbitmq
spec:
  selector:
    matchLabels:
      app: consumidor-rabbitmq
  template:
    metadata:
      labels:
        app: consumidor-rabbitmq
    spec:
      containers:
        - name: consumidor-rabbitmq
          image: ghcr.io/kedacore/rabbitmq-client:v1.0
          imagePullPolicy: Always
          command:
            - receive # Comando para recibir mensajes
          args:
            - "amqp://user:PASSWORD@rabbitmq.default.svc.cluster.local:5672" # URL de conexión a RabbitMQ
```

Este Deployment crea un pod que consume mensajes de la cola RabbitMQ. Utiliza la misma imagen de cliente de RabbitMQ y se conecta a la instancia de RabbitMQ utilizando las credenciales especificadas.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: consumidor-rabbitmq
  namespace: default
spec:
  scaleTargetRef:
    name: consumidor-rabbitmq # Referencia al despliegue del consumidor
  pollingInterval: 5 # Intervalo de sondeo en segundos
  cooldownPeriod: 30 # Periodo de enfriamiento en segundos
  maxReplicaCount: 30 # Número máximo de réplicas
  triggers:
  - type: rabbitmq
    metadata:
      queueName: hello # Nombre de la cola de RabbitMQ
      queueLength: "5" # Longitud de la cola para activar el escalado
    authenticationRef:
      name: autenticacion-consumidor-rabbitmq # Referencia a la autenticación
```

Este recurso define el ScaledObject de Keda, que permite el escalado automático del Deployment del consumidor basado en la longitud de la cola de RabbitMQ. La escala se realiza cada 5 segundos (pollingInterval). El número máximo de replicas es 30.

```
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: autenticacion-consumidor-rabbitmq
  namespace: default
spec:
  secretTargetRef:
  - parameter: host
    name: secreto-consumidor-rabbitmq # Referencia al secreto con las credenciales
    key: RabbitMqHost # Clave en el secreto que contiene la URL de RabbitMQ
```

Este recurso especifica las credenciales para acceder a RabbitMQ, referenciando el Secret creado anteriormente.

4.3 - Comparativa entre Keda y HPA

Durante la demostración, haré una breve comparativa entre Keda y HPA (Horizontal Pod Autoscaler), esta última es la solución que ofrece Kubernetes para solventar el problema del autoescalado en un cluster, es por eso que he visto necesario explicarlo en la memoria.

HPA (Horizontal Pod Autoscaler) es un recurso en Kubernetes que permite escalar automáticamente el número de réplicas de un conjunto de pods basado en métricas observadas como la utilización de CPU o memoria. La idea principal es mantener la disponibilidad de la aplicación mientras se optimizan los recursos utilizados.

Componentes Clave del HPA

Deployment o ReplicaSet: El HPA escala estos recursos. Generalmente, se configura para un Deployment que controla un conjunto de réplicas de pods.

Metric Server: Es necesario para obtener métricas del clúster. El servidor de métricas recopila y proporciona las métricas necesarias al HPA.

Controlador de HPA: Es un componente del control-plane de Kubernetes que revisa las métricas observadas y ajusta el número de réplicas en consecuencia.

Funcionamiento del HPA

- **Definición del HPA:** Se crea un recurso HPA que especifica la cantidad mínima y máxima de réplicas que puede tener el deployment, así como las métricas de destino (por ejemplo, 50% de utilización de CPU).
- **Monitoreo de Métricas:** El controlador de HPA consulta el Metric Server para obtener las métricas de los pods en el Deployment especificado.
- **Evaluación de las Métricas:** Compara las métricas obtenidas con los valores objetivo definidos en la configuración del HPA.
- **Aplicación del Cambio:** El HPA modifica el Deployment, ReplicaSet o StatefulSet correspondiente para ajustar el número de réplicas según lo calculado.

4.3.1 - Principales diferencias entre HPA y Keda

- TIPOS DE MÉTRICAS SOPORTADAS:

HPA :

Métricas Internas: Principalmente utiliza métricas de recursos internos, como la utilización de CPU y memoria de los pods.

Métricas Personalizadas: Soporta métricas personalizadas si se configuran adecuadamente usando Prometheus u otros servidores de métricas a través de la API de métricas personalizadas de Kubernetes.

Keda :

Métricas de Eventos: Está diseñado para trabajar con métricas basadas en eventos, como la longitud de colas en sistemas de mensajería, el consumo de mensajes, consultas de bases de datos, etc

Amplia Gama de Triggers: Soporta una amplia variedad de triggers basados en eventos, permitiendo el escalado en respuesta a diferentes tipos de eventos externos.

- DEPENDIENDO DEL ESCENARIO:

HPA:

Uso Común: Ideal para aplicaciones que necesitan escalar en función de la carga de trabajo típica, como sitios web, aplicaciones web o servicios que dependen de la utilización de CPU y memoria.

Métricas de Recursos: Escenarios donde el consumo de recursos (CPU, memoria) es el principal indicador de la necesidad de escalar.

Keda:

Escalado Basado en Eventos: Ideal para aplicaciones que necesitan escalar en respuesta a eventos específicos, como la llegada de mensajes a una cola, la aparición de nuevos archivos en un almacenamiento en la nube, o el cambio en métricas de servicios de nube.

Integraciones con Sistemas Externos: Escenarios donde el escalado debe ser impulsado por sistemas externos o métricas de eventos (colas de mensajes, logs, webhooks, etc.).

- CONFIGURACIÓN Y FACILIDAD DE USO DE CADA UNA:**HPA:**

Configuración Sencilla: Configuración relativamente sencilla usando recursos nativos de Kubernetes.

Limitaciones: Puede requerir configuración adicional para métricas personalizadas y puede no ser adecuado para todos los tipos de cargas basadas en eventos.

Keda:

Configuración Flexibilidad: Proporciona una configuración más flexible y específica para triggers de eventos.

Complemento: Actúa como un complemento para Kubernetes, lo que puede agregar una capa adicional de configuración y administración, pero también una mayor capacidad de respuesta a eventos.

5. Conclusiones y propuestas para seguir trabajando sobre el tema

Bajo mi punto de vista, Keda es una herramienta que puede llegar a ser muy útil e incluso indispensable a la hora de gestionar los recursos y la escalabilidad de nuestro clúster. Su capacidad para escalar pods basándose en una amplia variedad de eventos y métricas externas lo convierte en una solución muy útil para aplicaciones cuya carga de trabajo puede variar en pocos momentos. Mientras que HPA se centra en el escalado basado en métricas internas de recursos como CPU y memoria, Keda permite el escalado basado en eventos provenientes de servicios externos como colas de mensajes, bases de datos, y servicios en la nube.

POSIBLES MEJORAS O CONTINUACIONES

1. Monitoreo y Visibilidad Mejorados:

- Integración con Herramientas de Observabilidad: Incorporar integraciones más específicas con herramientas de monitoreo y observabilidad como Prometheus, Grafana, y otros sistemas para proporcionar una visibilidad completa del comportamiento del escalado.
- Alertas y Notificaciones: Configurar alertas y notificaciones basadas en eventos de escalado para mantener informados a los equipos de operaciones sobre los cambios en la infraestructura.

2. Automatización y Orquestación Avanzadas:

- Estrategias de escalado Personalizadas: como escalado basado en horarios o patrones de tráfico históricos.
- Integración con Flujos de Trabajo CI/CD: Automatizar la configuración y actualización de políticas de escalado como parte de los flujos de trabajo de CI/CD,

Me gustaría haber incorporado al proyecto la parte de alertas personalizadas a Slack o Correo electrónico cuando se realizase el autoescalado, pero no me ha dado tiempo a implementarlo.

Estuve haciendo varias pruebas con AlertManager, configurando un webhook para que las alertas fuesen enviadas a un grupo de trabajo de Slack, también probé Robusta, una herramienta de pago la cual tiene varios días de prueba gratis que consigue toda la parte de monitorización y alertas en tiempo real del clúster.

6. Dificultades que se han encontrado (optativo)

Para llegar a completar el TFG, me he encontrado con varias dificultades y cambios de que he tenido que solventar.

1. Creación del escenario:

- En primer lugar, mi idea era crear un escenario usando máquinas virtuales con Kubeadm, pero este método requiere de una cantidad de recursos de los que no dispongo en mi PC. También probé a crear el escenario en máquinas virtuales livianas, mi primera idea era automatizar la creación del escenario con Ansible y Vagrant, luego opté por otra alternativa ya que veía que esto me iba a tomar mas tiempo del esperado, además para el escenario y las pruebas que voy a hacer, necesito un entorno dinámico el cual pueda eliminar y levantar en cuestión de segundos.

- Como segunda opción, opté por montar el cluster con máquinas virtuales pero esta vez usando K0S, esta es una distribución de Kubernetes la cual permite crear y gestionar de forma ligera, eficiente y fácil de usar un clúster de Kubernetes.

Esta distribución funciona con un binario el cual tenemos que descargar en nuestro equipo anfitrión, para lanzar la instalación debemos tener acceso por SSH a las máquinas virtuales que voy a utilizar, esta instalación puede ser mediante un fichero o comandos sencillos.

El problema aquí fue que la configuración no me llegó a funcionar, usé máquinas vagrant a las cuales tengo acceso por SSH, aún así no pude levantar el escenario con esta herramienta si el cluster contaba con mas de 1 control-plane.

- Pensé en crear el escenario en máquinas virtuales con k3s, ya que esta era una herramienta que se ha utilizado en clase y es muy sencilla de usar para crear los clústers, pero quise buscar algo que no se hubiese visto en clase.

- Finalmente encontré Kind, el cual permite crear clústers de prueba con un consumo mínimo de recursos y de una forma extremadamente rápida y sencilla, lo cual era exactamente lo que necesitaba para el proyecto.

2. Familiarización con Kind

Ya tenía claro que Kind era la herramienta que usaría para la creación del escenario, aunque esta presentaba algunos inconvenientes, como su gran limitación en el rendimiento ya que ocupa muy pocos recursos, también encontré complejidades leves en la configuración de red en Kind.

3. Implementación de las aplicaciones

La primer dificultad que encontré a la hora de implementar las aplicaciones, fue saber que fuente de datos usar, ya que Keda es compatible con una gran cantidad de métricas externas para el autoescalado. Opté por las que me resultaron más llamativas. Otra de las dificultades que tuve llegó a la hora de buscar información en la implementación de estas aplicaciones con Keda. La única que encontré fue con MongoDB, esta la tomé de ejemplo para desarrollar las demás.

7. Bibliografía

<https://github.com/kedacore/samples?tab=readme-ov-file>

<https://doc.kaas.thalesdigital.io/docs/Features/keda>

<https://www.nearform.com/insights/autoscaling-kubernetes-with-keda/>

<https://klaushofrichter.medium.com/autoscaling-with-keda-70e5b12be492>

<https://www.armosec.io/platform/kubernetes-security-posture-management/>

<https://keda.sh/>

<https://medium.com/cuddle-ai/auto-scaling-microservices-with-kubernetes-event-driven-autoscaler-keda-8db6c301b18>

<https://devtron.ai/blog/introduction-to-kubernetes-event-driven-autoscaling-keda/>

<https://medium.com/@carlocolumna/how-to-scale-your-pods-based-on-http-traffic-d58221d5e7f1>

<https://blog.devops.dev/keda-autoscaling-kubernetes-apps-using-prometheus-da037fe572cf>

<https://engineering.intility.com/article/scaling-kubernetes-apps-with-keda>

<https://isitobservable.io/observability/kubernetes/introducing-event-driven-autoscaling-with-keda>

<https://www.spectrocloud.com/blog/kubernetes-autoscaling-patterns-hpa-vpa-and-keda>