COMPTE_RENDU_CRV_OULFID_VARATHARAJAH

KUBERNETES

Ce projet a pour objectif de mettre en place une infrastructure Kubernetes permettant de déployer un ensemble de services applicatifs tout en assurant leur monitoring avec Prometheus et Grafana. L'application repose sur un serveur backend, un frontend et une base de données Redis.

L'architecture du projet repose sur plusieurs composants principaux :

- **Kubernetes** : orchestration des conteneurs et gestion des services.
- **Docker** : conteneurisation des différentes applications.
- **Prometheus** : collecte et stockage des métriques des services.
- **Grafana** : visualisation des métriques sous forme de tableaux de bord.
- **Redis** : base de données en mémoire.
- **Node.js** : serveur backend de l'application.

1.Creation des Images

Il a fallut dans un premier temps créer les images du projet pour cela nous avons utiliser docker pour créer une image pour la parti node-redis et la partie front .
L'image redis étant déjà fourni par redis pas besoin de la build.

```
docker build
```

Puis on a push ces images sur notre docker hub.

```
docker push
```

2.Mise en place de Kubernetes

Pour la partie Kubernetes nous avons d'abord créer nos 3 déploiements qui sont :
- serveur-deployment -> 3 réplicas
- serveur-service exposé en 3001 en LoadBalancer
- redis-deployment -> contient 1 réplicas
- redis-service n'est pas exposé vers l'extérieur accessible depuis le port 6379 en ClusterIP

- frontend-front-deployment -> contient 1 réplicas
- front-service exposé sur le port 80 en LoadBalancer

Un fois les fichier crée il suffit d'apply :

```
kubectl apply -f serveur-deployment.yml
kubectl apply -f redis-deployment.yml
kubectl apply -f front-deployment.yml
```

Puis apres cela nous avons modifier le fichier conf.js qui est dans le folder redis-react:
qui permet de dire au front qu'elle serveur contater pour traiter les requêtes:
Non avons donc mis bblabla

Cependant, l'IP publique de ce service change chaque fois que nous redémarrons les pods (cela peut être causé par un redémarrage de Minikube, la mise à jour des ressources, ou d'autres facteurs). Le problème est que, dans le front-end de l'application, l'IP du backend est hardcodée, ce qui signifie qu'il nous faudrait constamment mettre à jour cette adresse IP chaque fois que celle-ci change.

Pour résoudre ce problème, nous avons utilisé une variable d'environnement dans notre application React (front-end). Au lieu de coder en dur l'IP du backend dans notre code JavaScript, nous avons utilisé la variable d'environnement REACT_APP_API_URL dans le fichier de configuration de notre projet.
Cela nous permet de modifier dynamiquement l'URL de l'API sans avoir à recompiler ou modifier le code source.

```
export const URL = process.env.REACT_APP_API_URL
```

Puisque l'IP du backend est dynamique, nous avons besoin d'un moyen automatique pour mettre à jour cette valeur dans l'environnement de l'application front-end chaque fois que l'IP change.

Pour ce faire, nous avons utilisé un ConfigMap dans Kubernetes. Un ConfigMap est une ressource Kubernetes qui permet de stocker des configurations sous forme de paires clé-valeur (par exemple, REACT_APP_API_URL). Ce ConfigMap est ensuite utilisé dans notre déploiement de frontend** pour injecter cette configuration dans les conteneurs de l'application React via une variable d'environnement.

---

front-configmap.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: frontend-config
data:
  REACT_APP_API_URL: ""  # L'IP du backend (dynamique)
```

Le problème suivant est de récupérer l'IP dynamique du service Kubernetes (qui est exposé via un LoadBalancer en utilisant Minikube). Pour ce faire, nous avons créé un script Bash qui exécute la commande kubectl get service pour obtenir l'IP et le port du service backend, puis met à jour le ConfigMap avec cette nouvelle IP. Cela permet à l'application front-end d'obtenir l'IP la plus récente à chaque redémarrage des pods.

update-configmaps.sh

Ci dessous les lignes importantes du script

```bash
# Récupérer l'IP du service backend
NEW_IP=$(kubectl get svc serveur-service -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')

# Mettre à jour le ConfigMap avec la nouvelle IP
kubectl create configmap frontend-config --from-
literal=REACT_APP_API_URL="http://$NEW_IP:3001" --dry-run=client -o yaml |
kubectl apply -f -
```

Ce script est exécuté après le démarrage de Minikube et lors du redémarrage des pods, afin de s'assurer que l'IP du backend est toujours à jour dans la configuration du front-end. Cela garantit que, peu importe la manière dont l'IP change, l'application front-end utilise toujours la bonne IP pour se connecter au backend.

L'option **-o jsonpath='{.status.loadBalancer.ingress[0].ip}'** extrait uniquement l'IP depuis le JSON retourné par Kubernetes.

--from-literal=REACT_APP_API_URL="http://$NEW_IP:3001" : Crée la variable `REACT_APP_API_URL` contenant l'IP du backend.

| kubectl apply -f - : Applique la mise à jour du ConfigMap sans le recréer (sinon il faut le supprimer avant).

A ce niveau du projet nous avons donc 4 fichier qui sont :

- serveur-deployment.yml
- front-deployment.yml
- redis-deployment.yml
- frontend-configmap.yml

Services:

```
hamza@hamza-VirtualBox:~/Projet1_CRV/Kubernetes$ kubectl get service
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)          AGE
front-service       LoadBalancer   10.105.116.81    10.105.116.81    80:30900/TCP     3h
grafana             NodePort       10.97.184.87     <none>           3000:30000/TCP   167m
kubernetes          ClusterIP      10.96.0.1        <none>           443/TCP          3h2m
prometheus-service  NodePort       10.102.111.117   <none>           9090:30263/TCP   177m
redis-exporter      ClusterIP      10.111.133.88    <none>           9121/TCP         177m
redis-service       ClusterIP      10.101.155.90    <none>           6379/TCP         3h
serveur-service     LoadBalancer   10.102.225.108   10.102.225.108   3001:32705/TCP   3h
```

Deployments:

```
hamza@hamza-VirtualBox:~/Projet1_CRV/Kubernetes$ kubectl get deployments
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
front-deployment           1/1     1            1           3h2m
grafana                    1/1     1            1           168m
prometheus                 1/1     1            1           179m
redis-deployment           1/1     1            1           3h2m
redis-exporter             1/1     1            1           179m
redis-replica-deployment   6/6     6            6           3h2m
serveur-deployment         3/3     3            3           3h2m
```

Dans un seconde temps on a crée le replica de redis-deployment.
L'objectif de ce déploiement était de mettre en place Redis avec une architecture maître-réplicas dans Kubernetes. Cela signifie qu'on a un serveur Redis principal (master) qui accepte les écritures et plusieurs réplicas qui reçoivent les mises à jour du maître en lecture seule.
Extrait du code qui permet cela

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-replica-deployment
spec:
  replicas: 2  # Nombre de réplicas
  selector:
    matchLabels:
      app: redis-replica
  template:
    metadata:
      labels:
        app: redis-replica
    spec:
      containers:
        - name: redis-replica
          image: redis:7
          ports:
            - containerPort: 6379
          command: ["redis-server", "--replicaof", "redis-service", "6379"]
```

Cela signifie que ces instances vont se synchroniser avec le maître situé à l'adresse redis-service sur le port 6379.

Par la suite nous avons créer des script qui permettent d'automatiser la mise en place de notre infrastructure de A-Z.

./Tunnel.sh: Permet de lancer minikube et de lancer le tunnel qui nous permet d'avoir des IP pour les LoadBalancer

./Start.sh: Permet de créer les deployement dans un ordre précis afin d'eviter les erreurs et aussi de mettre a jour l'IP de notre variable d'environnement en utilisant ***update-configmaps.sh***.

```
hamza@hamza-VirtualBox:~/Projet1_CRV/Kubernetes$ kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
front-deployment-56967779d8-qxfns        1/1     Running   0          179m
grafana-755fd46679-cbmhp                 1/1     Running   0          155m
prometheus-85b5b98fdd-bmxm8              1/1     Running   0          177m
redis-deployment-66998d46cc-d6gkd        1/1     Running   0          3h
redis-exporter-cc4bc85cf-kbbdw           1/1     Running   0          177m
redis-replica-deployment-6d9cc5b8d7-4knvz   1/1     Running   0          3h
redis-replica-deployment-6d9cc5b8d7-9qpg5   1/1     Running   0          3h
redis-replica-deployment-6d9cc5b8d7-f75bt   1/1     Running   0          3h
redis-replica-deployment-6d9cc5b8d7-llxjx   1/1     Running   0          3h
redis-replica-deployment-6d9cc5b8d7-m6tmh   1/1     Running   0          3h
redis-replica-deployment-6d9cc5b8d7-npq8r   1/1     Running   0          3h
serveur-deployment-b55b57844-9pf6s       1/1     Running   0          3h
serveur-deployment-b55b57844-v5qnd       1/1     Running   0          3h
serveur-deployment-b55b57844-vjffc       1/1     Running   0          3h
```

AUTOSCALING DU BACKEND SERVEUR

L'autoscaling (ou mise à l'échelle automatique) du backend permet d'adapter dynamiquement le nombre de pods en fonction de la charge de travail (CPU ou mémoire). Cela garantit une meilleure résilience, scalabilité et performance du système : si le backend reçoit beaucoup de requêtes, Kubernetes crée automatiquement de nouveaux pods pour absorber la charge ; à l'inverse, il les réduit quand l'activité diminue, ce qui optimise l'usage des ressources.
Pour activer cette fonctionnalité, plusieurs fichiers et éléments ont été mis en place.

1)metrics-server.yml:

Ce composant est essentiel car c'est lui qui fournit les métriques de ressources (CPU, mémoire, etc.) utilisées par Kubernetes pour prendre des décisions d'autoscaling. Il doit être déployé dans le cluster pour que le HPA (Horizontal Pod Autoscaler) fonctionne. Le fichier metrics-server.yml est issue une ressource officielle fournie par la team Kubernetes. On a modifié ce fichier localement, car par défaut il nécessite un certificat valide et un accès réseau bien configuré, on a ici rajouté la ligne "--kublet-insecure-tls".

```
spec:
  containers:
    - args:
        - --cert-dir=/tmp
        - --secure-port=10250
        - --kubelet-insecure-tls
```

2)horizontalpod.yml:

Ce fichier définit une ressource de type HorizontalPodAutoscaler. Elle permet de surveiller un déploiement donné (ici le backend serveur-deployment) et d'ajuster automatiquement le nombre de pods selon l'utilisation du CPU.

```
metadata:
  name: serveur-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: serveur-deployment
  minReplicas: 2
  maxReplicas: 6
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

Cela signifie que si l'utilisation moyenne du CPU dépasse 50 %, Kubernetes va progressivement augmenter le nombre de pods (jusqu'à 6 dans notre fichier). Si l'utilisation redescend, les pods seront supprimés (mais jamais moins de 2 dans ce cas).

3)serveur-deployment.yml:

Ce fichier contient la définition du backend. Il doit spécifier les limites et les requêtes CPU dans le conteneur, car c'est à partir de ces valeurs que le HPA calcule le pourcentage d'utilisation.

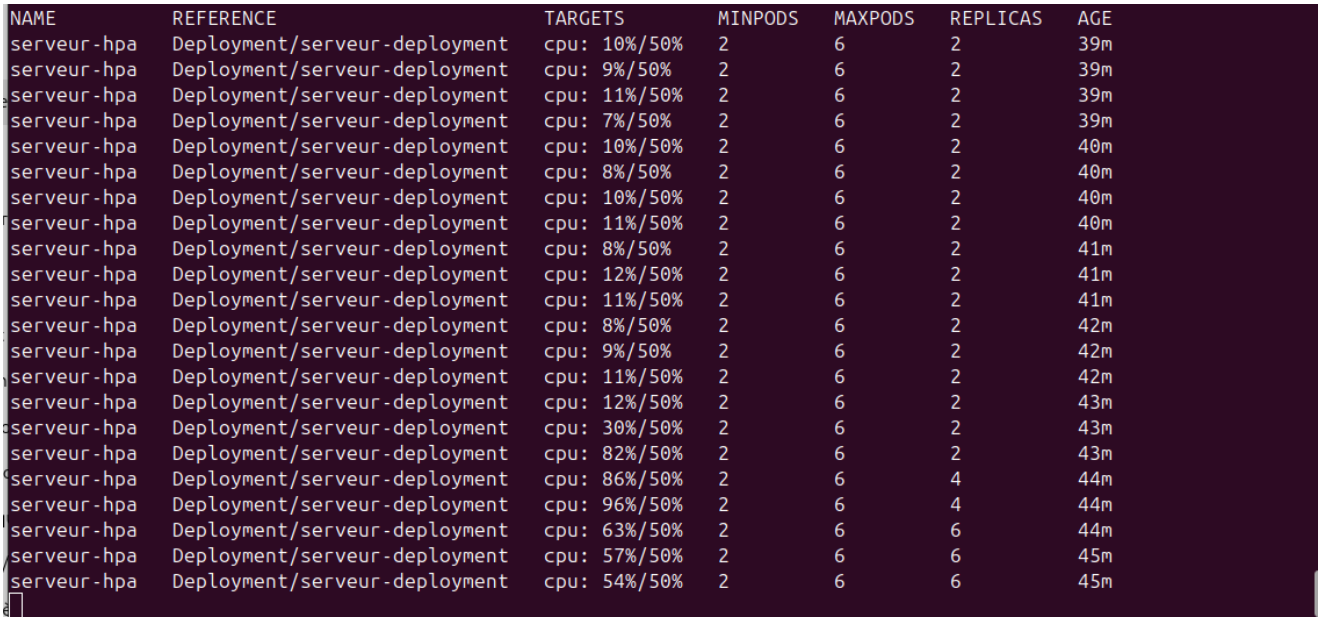
```
resources:
  requests:
    cpu: "100m"
    memory: "128Mi"
  limits:
```

```
      cpu: "500m"
      memory: "256Mi"
```

Cette section définit les ressources minimales (requests) et maximales (limits) que le pod peut utiliser. Requests garantit que le pod aura au moins 100m de CPU et 128Mi de mémoire et Limits empêche le pod de dépasser 500m de CPU et 256Mi de mémoire, même s'il en demande plus.

Utilisation et exemples:
Kubectl apply –f metrics-serveur.yml
Kubectl apply –f horizontalpod.yml
Kubectl apply –f serveur-deployment.yml
Dans un terminal on lance
kubectl get hpa –A
Puis dans un autre on lance le script, stress-backend.sh :
./stress-backend.sh

Et enfin on observe :



NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
serveur-hpa	Deployment/serveur-deployment	cpu: 10%/50%	2	6	2	39m
serveur-hpa	Deployment/serveur-deployment	cpu: 9%/50%	2	6	2	39m
serveur-hpa	Deployment/serveur-deployment	cpu: 11%/50%	2	6	2	39m
serveur-hpa	Deployment/serveur-deployment	cpu: 7%/50%	2	6	2	39m
serveur-hpa	Deployment/serveur-deployment	cpu: 10%/50%	2	6	2	40m
serveur-hpa	Deployment/serveur-deployment	cpu: 8%/50%	2	6	2	40m
serveur-hpa	Deployment/serveur-deployment	cpu: 10%/50%	2	6	2	40m
serveur-hpa	Deployment/serveur-deployment	cpu: 11%/50%	2	6	2	40m
serveur-hpa	Deployment/serveur-deployment	cpu: 8%/50%	2	6	2	41m
serveur-hpa	Deployment/serveur-deployment	cpu: 12%/50%	2	6	2	41m
serveur-hpa	Deployment/serveur-deployment	cpu: 11%/50%	2	6	2	41m
serveur-hpa	Deployment/serveur-deployment	cpu: 8%/50%	2	6	2	42m
serveur-hpa	Deployment/serveur-deployment	cpu: 9%/50%	2	6	2	42m
serveur-hpa	Deployment/serveur-deployment	cpu: 11%/50%	2	6	2	42m
serveur-hpa	Deployment/serveur-deployment	cpu: 12%/50%	2	6	2	43m
serveur-hpa	Deployment/serveur-deployment	cpu: 30%/50%	2	6	2	43m
serveur-hpa	Deployment/serveur-deployment	cpu: 82%/50%	2	6	2	43m
serveur-hpa	Deployment/serveur-deployment	cpu: 86%/50%	2	6	4	44m
serveur-hpa	Deployment/serveur-deployment	cpu: 96%/50%	2	6	4	44m
serveur-hpa	Deployment/serveur-deployment	cpu: 63%/50%	2	6	6	44m
serveur-hpa	Deployment/serveur-deployment	cpu: 57%/50%	2	6	6	45m
serveur-hpa	Deployment/serveur-deployment	cpu: 54%/50%	2	6	6	45m

PROMETHEUS ET GRAFANA

Afin de mettre en place notre outil de monitoring prometheus nous avons creer un fichier **prometheus-deployment.yml**.
Ce fichier YAML définit le déploiement de Prometheus, un outil de monitoring, ainsi que de Redis Exporter, qui expose les métriques de Redis pour Prometheus.

Déploiement de Prometheus:
Il utilise l'image officielle prom/prometheus:latest et charge une configuration personnalisée via un ConfigMap. Prometheus écoute sur le port 9090.

ConfigMap – prometheus-config
Il contient le fichier prometheus.yml avec deux scrape_configs :
- nodejs pour récupérer les métriques du backend Node.js via serveur-service:3001/metrics
- redis pour récupérer les métriques via redis-exporter:9121/metrics

Service Prometheus
Expose Prometheus via un service de type NodePort pour rendre l'interface web accessible en dehors du cluster.

Déploiement de Redis Exporter
Utilise l'image officielle oliver006/redis_exporter, configurée pour interroger Redis grâce à la variable d'environnement REDIS_ADDR.

Service Redis Exporter
Permet à Prometheus d'accéder aux métriques exposées par Redis Exporter sur le port 9121.

Puis par la suite nous avons mis en place grafana qui est est un outil de visualisation qui permet de créer des tableaux de bord interactifs à partir de métriques collectées.

Pour cela on a crée un fichier **graphana.yml** :

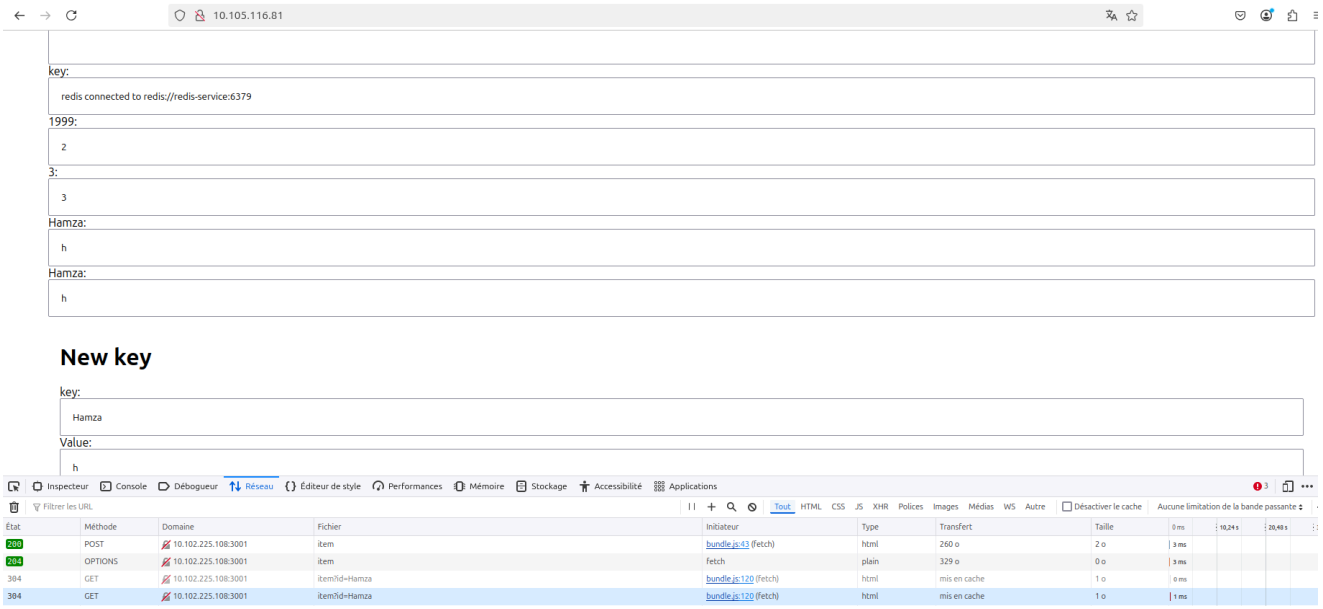
Déploiement de Redis Exporter
Utilise l'image officielle grafana/grafana:latest.

Service grafana
Expose Grafana via un service de type NodePort pour rendre l'interface web accessible en dehors du cluster en port 3000.

Un fois que graphan est deployé il suffit de creer une nouvelle data base a partir de l'URL de notre service promtheus puis ensuite de choisir quelle metric on souhaite visualiser.

On peut voir que en rafraichissant la page et en faisant des requêtes post la courbe en bleu augmente.


Grafana
Home
Bookmarks
Starred
Dashboards
Playlists
Snapshots
Library panels
Shared dashboards
Explore
Drilldown
New!
Alerting
Connections
Administration
Home › Dashboards › Kubernetes cluster monitoring (via Prometheus) › View panel
Search or jump to...
Back to dashboard
Save dashboard
Node
All
Last 5 minutes
Refresh
10s
Cluster memory usage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
10:05:45
10:06:00
10:06:15
10:06:30
10:06:45
10:07:00
10:07:15
10:07:30
10:07:45
10:08:00
10:08:15
10:08:30
10:08:45
10:09:00
10:09:15
10:09:30
10:09:45
10:10:00
10:10:15
10:10:30
{__name__="redis_commands_total", cmd="client|setinfo", instance="redis-exporter.default.svc.cluster.local:9121", job="redis"}
{__name__="redis_commands_total", cmd="client|setname", instance="redis-exporter.default.svc.cluster.local:9121", job="redis"}
{__name__="redis_commands_total", cmd="config|get", instance="redis-exporter.default.svc.cluster.local:9121", job="redis"}
{__name__="redis_commands_total", cmd="get", instance="redis-exporter.default.svc.cluster.local:9121", job="redis"}


10.105.116.81
key:
redis connected to redis://redis-service:6379
1999:
2
3:
3
Hamza:
h
Hamza:
h
New key
key:
Hamza
Value:
h
Inspecteur
Console
Débogueur
Réseau
Éditeur de style
Performances
Mémoire
Stockage
Accessibilité
Applications
Filtrer les URL
Tout
HTML
CSS
JS
XHR
Polices
Images
Médias
WS
Autre
Désactiver le cache
Aucune limitation de la bande passante
État
Méthode
Domaine
Fichier
Initiateur
Type
Transfert
Taille
200
POST
10.102.225.108:3001
item
bundle.js:43 (fetch)
html
260 o
2 o
204
OPTIONS
10.102.225.108:3001
item
fetch
plain
329 o
0 o
304
GET
10.102.225.108:3001
item?id=Hamza
bundle.js:120 (fetch)
html
mis en cache
1 o
304
GET
10.102.225.108:3001
item?id=Hamza
bundle.js:120 (fetch)
html
mis en cache
1 o